VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

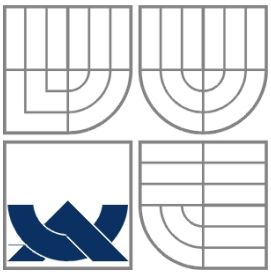
GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

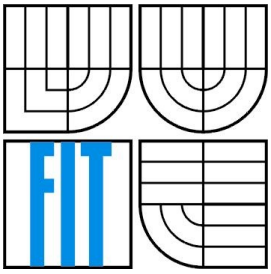
AUTOR PRÁCE
AUTHOR

Bc. MARKÉTA DUBSKÁ

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

GRAPHICS INTRO 64KB USING OPENGL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARKÉTA DUBSKÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ADAM HEROUT, Ph.D.

BRNO 2010

Abstrakt

Témou diplomovej práce je tvorba grafického intra s veľkosťou do 64kB. V nasledujúcom texte je stručne uvedená história demoscény a grafická knižnica OpenGL. Hlavnou časťou je však popis minimalistických techník použitých pri tvorbe a ich praktické využitie vo výslednom programe.

Abstract

The aim of the master's thesis is to create a graphic intro with the limited size of 64kB. Brief history of a demoscene and the graphics library OpenGL is introduced in the following text. The main part of the work is a description of minimal graphics techniques used in a creative process and their further application in the final programme.

Klíčová slova

demo, intro 64kB, Perlinov šum, celulárne textúry, bump mapping, algoritmus prerozdelenia Catmull-Clark, Bézierove krivky, shadre, zvukový syntetizér, výšková mapa, tiene

Keywords

demo, intro 64kB, Perlin Noise, cellular textures, bump mapping, Catmull-Clark subdivision, Bézier curves, shaders, sound sythesizer, elevation map, shadows

Citace

Markéta Dubská: Grafické intro 64kB s použitím OpenGL, diplomová práce, Brno, FIT VUT v Brně, 2010

Grafické intro 64kB s použitím OpenGL

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením Ing. Adama Herouta, Ph.D.
Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

……
Markéta Dubská
25. 5. 2010

Poděkování

Rada by som poďakovala vedúcemu mojej diplomovej práce Ing. Adamovi Heroutovi, Ph.D. za odborné vedenie.

Obsah

1 Úvod

Pokrok vo vývoji hardwaru a softwaru je v dnešnej dobe nezadržateľný. Pred niekoľkými rokmi stačilo pre uchovanie dát niekoľko megabytov. Dnes už nie sú ničím neobvyklým ani terabytové disky. Z tohto dôvodu by sa mohlo zdať zbytočné vytvárať programy s minimálnou veľkosťou. Je však potrebné položiť si otázku, či má aplikácia zaberať stovky alebo desiatky MB. Hoci demá[1] a intrá[2] nie sú programy, ktoré by sa používali pre ich funkčnosť, sú ukážkou programátorských schopností autora a mnohé použité algoritmy môžu byť aplikované vo väčších projektoch. Súčasne sú ukážkou hardwarových možností grafických kariet a tým posúvajú hranice ich ďalšieho využitia.

Hoci sa zdá byť veľkosť veľmi limitujúca, vybrané diela od najvýznamnejších tvorcov sú dôkazom toho, ako môže byť týchto 65 536 bytov dokonale využitých. V nasledujúcich kapitolách sú priblížené techniky, ktoré boli v práci využité a problémy, ktoré sa pri vytváraní projektu objavili.

V prvej kapitole je uvedená história a súčasnosť demoscény. Keďže bola pri tvorbe využitá knižnica OpenGL, je stručne popísaná v druhej kapitole. Dôležitým faktorom pre výslednú veľkosť programu je správne nastavenie kompilátora, ktorému sa venuje štvrtá kapitola. Použité matematické funkcie sú priblížené v nasledujúcej kapitole. Šiesta kapitola popisuje použité techniky pre tvorbu modelov, textúr a efektov. Obsahuje podkapitoly *Textúry*, *3D objekty*, *Tiene*, *Point sprity*, *Stencil test*, *Fonty*, *Výšková mapa*, *Obloha* a *Krivky a splajny*. Určitá časť výsledného programu je napísaná v programovacom jazyku GLSL. Viac o tomto jazyku a jeho použití pojednáva kapitola *GLSL a shadre*. Neoddeliteľnou súčasťou grafického intra je hudobný doprovod a pohyb kamery. Tejto téme sa venujú nasledujúce dve kapitoly. Záver práce obsahuje zhrnutie výsledného programu, kompresiu a porovnania behu programu pri rôznych nastaveniach a použitých grafických kartách.

1 demo - z angl. *demonstrate* - ukázať, predviesť a pod.

2 intro - z angl. *introduction* - úvod, uvedenie, predstavenie a pod.

2 História a súčasnosť demoscény

2.1 Demoscéna

Počiatok demoscény sa datuje okolo roku 1980. Vznikla s príchodom prvých komerčných hier pre 8-bitové domáce počítače, v tej dobe napr. *Commodore 64* alebo *Amiga*. Nie každý bol ochotný hru zakúpiť, a tak vznikla komunita ľudí, ktorá sa snažila odstrániť z hier ochrany proti kopírovaniu (tzv. hru „*cracknúť*"[3]). Zákonite sa objavila rivalita medzi jednotlivými skupinami, snažiacimi sa prelomiť jednotlivé hry. Ako podpis vkladali pred programy svoje malé „cractrá". Toto pomenovanie vzniklo spojením slov *crack* a *intro*. Tie obsahovali jednoduché grafické efekty, nápisy, prípadne odkaz ostatným skupinám. Časom sa intrá úplne oddelili od hier a vznikali ako samostatné programy.

S masovým príchodom osobných počítačov sa demá a intrá rozšírili postupne aj na platformy ako DOS a v dnešnej dobe najrozšírenejšie - Windows a Linux.

Dnes sa dá demoscéna charakterizovať ako celosvetová, nekomerčná skupina združujúca kreatívnych ľudí so záľubou v tvorbe dem. Patria medzi nich nielen programátori, ale aj grafici, hudobníci či scenáristi. Tí sa združujú do demoskupín, v ktorých spoločne tvoria výsledný produkt. Medzi najslávnejšie a najúspešnejšie demoskupiny patria nemecký *Farbraush* [18], maďarský *Conspiracy*, či medzinárodný *RGBA* [19].

V začiatkoch prezentovali demoskupiny svoje diela nelegálne, a to vložením do danej hry. Po oddelení od hier si autori programy posielali poštou na disketách a stretávali sa na malých tzv. demopárty. S príchodom internetu sa značne rozšírili možnosti demá a intrá publikovať. Dnes patria medzi najznámejšie portály *scene.org* a *pouet.net*, kde je možné diela nielen prezentovať, ale aj získať mnoho informácií pre písanie kódu a diskutovať na rôzne témy týkajúce sa demoscény a samotných diel. Takisto sa zmenili aj demopárty. Počiatočné stretnutia pár ľudí sa rozvinuli na veľmi dobre organizované medzinárodné akcie, ktorých súčasťou bývajú aj semináre a diskusie o technikách tvorby dem. Hlavným ceiľom však naďalej zostávajú súťaže o najlepšie dielo v danej kategórií. Medzi najväčšie demopárty možno zaradiť *Breakpoint*, ktorý sa každoročne koná v Nemecku a fínsku demopárty *Assembly*. Cena pre víťaza môže byť symbolická alebo aj v hodnote niekoľko tisíc eur, v závislosti od prestíže súťaže. Najväčšou odmenou je však pre víťaza uznanie od konkurencie.

3 cracknúť - z angl. *crack* - prelomiť, rozlúsknuť

2.2 Demo a intro

Výsledný program sa podľa svojej veľkosti zaraďuje do rôznych kategórií. Medzi spoločné charakteristiky môžeme zaradiť neinteraktivitu a snahu vytvoriť čo najkomplexnejšie audiovizuálne dielo, ktoré sa snaží maximálne využiť hardwarové možnosti počítačov. Ku hlavným kategóriám patria najmä demo, 64K intro, 4K intro a 256B intro. Demá mávajú veľkosť niekoľko megabytov, pričom maximálna veľkosť je zväčša určená podmienkami demopárty, na ktorej je demo zverejnené. Na rozdiel od dem, majú intrá veľkosť obmedzenú, ako už názvy napovedajú, na 65 536B, 4 096B či dokonca len 256B. Najväčším problémom tvorby intra je ich hardwarová závislosť. Hoci sa snažia byť v maximálnej možnej miere prenositeľné, kvôli rôznym technickým parametrom grafických kariet je túto podmienku náročné splniť.

V minulosti sa ako hlavný programovací jazyk používal assembler. Dnes sa už bežne programuje vo vyšších jazykoch ako C/C++. Pre tvorbu 4K a 256B intier sa však stále používa assembler, pomocou ktorého je možné dosiahnuť najväčšiu úsporu použitého priestoru. Kategória do 256B sa líši od ostatných aj chýbajúcim hudobným doprovodom. Zdrojové kódy výsledných programov si jednotlivé skupiny dôsledne chránia, a preto je náročné nájsť presne implementované algoritmy. Existuje však veľa publikácií, kde sú tieto minimalistické techniky popísané. Samotné naprogramovanie ostáva na autorovi. Niektoré demoskupiny však sprístupnili pre voľné použitie vlastné nástroje a knižnice, ktoré pri tvorbe používajú. Medzi takéto nástroje patrí napríklad knižnica *V2 Synthesizer System* [21] pre import hudobného doprovodu a program *kkrunchy* [22] vhodný na kompresiu výsledného súboru, ktorý bol použitý aj v tejto práci.

3 Vývojové prostredie

3.1 OpenGL

Ako už názov práce napovedá, výsledný program je vytvorený za pomoci knižnice OpenGL (*Open Graphics Library*). Tá tvorí aplikačne rozhranie ku grafickým kartám a subsystémom [5]. Spolu s Direct3D z produkcie Microsoftu dnes predstavuje štandard pre tvorbu 2D a 3D grafických aplikácií. Prvá verzia pochádza z roku 1992 od firmy SGI (*Silicon Graphics Inc.*) a odvtedy sa neustále vyvíja. Knižnica sa snaží využívať grafické akcelerátory a v prípade, že nie sú nainštalované, použije sa softwarová simulácia. Aktuálna verzia 4.0 z 10. marca 2010 poskytuje široké pole funkcií pre prácu s geometriou či textúrami. Od verzie 2.0 je obsahom štandardu aj vlastný shadrovací jazyk GLSL, ktorý umožňuje zásah do fixnej pipeline vykresľovacieho reťazca OpenGL. Táto verzia navyše odstraňuje obmedzenie rozmerov textúr na mocninu dvoch a umožňuje použitie bodových spritov s textúrou. Verzia 3.0 sa od predchádzajúcej líši napríklad zavedením vertexových polí, plnou podporou renderovania do objektu framebufferu, či pridaním nových kompresných možností.

Pre OpenGL existujú rôzne implementácie, ktoré ponúkajú využitie knižnice prakticky na všetkých dnes rozšírených platformách. Rozhranie knižnice je navrhnuté tak, aby ju bolo možno použiť v spojení s ľubovoľným programovacím jazykom. Dnes sa používa najmä v spojení s C/C++.

Pri tvorbe aplikácií s použitím OpenGL sa široko využívajú rozširujúce knižnice ako je *GLU*[4] alebo *GLUT*[5]. Tie umožňujú vykreslenie základných geometrických objektov ako kocka či valec, alebo poskytujú funkcie pre vytvorenie okna aplikácie a zachytávanie udalostí. Ďalšou pomerne používanou je knižnica *GLEW*[6] pre prácu s rozšíreniami. Použitie týchto knižníc samozrejme zvyšuje výslednú veľkosť programu. Z tohto dôvodu je ich integrovanie do výsledného programu minimálne. Pripojenie rozšírení, ktoré zabezpečuje knižnica *GLEW* je implementované pomocou vlastných funkcií. Knižnica *GLUT* nie je použitá vôbec a z knižnice *GLU* sú použité funkcie pre nastavenie projektívnej a modelovo-pohľadovej matice ako sú `gluOrtho2D()`, `gluLookAt()` a `gluPerspective()`.

4 GLU (*OpenGL Utility Library*) - rozširujúca knižnica pre prácu s kvadrikami, mipmapami, krivkami, tesaláciu polygónov,...

5 GLUT (*OpenGL Utility Toolkit*) - knižnica, ktorá obsahuje nástroje pre správu okna a zachytávanie udalostí a vykresľovanie základných objektov ako kocka alebo guľa.

6 GLEW (*OpenGL Extension Wrangler Library*) - knižnica pre prácu s rozšíreniami OpenGL.

3.2 Vývojové prostredie

Na začiatku vývoja každej aplikácie sa programátor musí rozhodnúť, aké vývojové prostredie zvoliť či aký kompilátor použiť. Pri tvorbe menších projektov, kde sa nekladú špeciálne požiadavky na veľkosť či iné parametre výsledného programu, je toto rozhodnutie pomerne jednoduché. Jeho voľba závisí výlučne na programátorovi, ktorý si zvolí pre svoju prácu pre neho najvhodnejšie vývojové prostredie. Pri tvorbe dema je však správne nastavenie kompilátora a linkera jednou z kľúčových úloh. Tá istá aplikácia môže dosahovať pri rôzne zvolených prepínačoch niekoľkonásobne rozdielnu veľkosť.

Možnosťou je napísanie programu v jednoduchom textovo editore a následná kompilácia, napríklad pomocou *MinGW*[7]. Nevýhodou je malá podpora zvýraznenia syntaxi či chýbajúca nápoveda, čo môže byť u väčších projektov obmedzujúce. Preto je vhodné použiť vývojové prostredie uľahčujúce prácu a orientáciu medzi jednotlivými súbormi. Medzi možné voľby patrí napríklad *Dev-C++* vyvinutý firmou *Bloodshed Software* používajúci spomínanú sadu kompilátorov *MinGW*. Jedným z najrozšírenejších programov pre vývoj aplikácií je prostredie *Microsoft Visual Studio* (ďalej len *VS*) od firmy *Microsoft*. Veľa tutoriálov a ukážkových programov na internete sú práve projekty vytvorené vo *VS*. Toto vývojové prostredie bolo zvolené aj pre potreby tejto práce, a to jeho špecifikácia *Visual C++*. Súčasťou *VS* je mnoho nástrojov pre podporu vývoja projektov, ako napríklad vlastný debugger a kompilátor. Široké možnosti parametrov kompilátora a prehľadné rozhranie umožňuje veľmi dobrú kontrolu nad optimálnymi nastaveniami.

Pri vývoji aplikácie boli definované dve nastavenia pre preklad programu:

- *debug* – nastavenie pre ladenie programu, odchyt udalostí, generovanie výstupného súboru s informáciami zaznamenanými počas behu a pod.
- *release* – verzia pre finálnu podobu dema so všetkými potrebnými nastaveniami linkera a prepínačov pre minimalizáciu výsledného programu. Tieto parametre sú bližšie popísané v nasledujúcich podkapitolách.

Nastavenia prepínačov sú popísané pre rozhranie *VS*, avšak pre iné kompilátory sú spomínané parametre identické alebo podobné.

3.3 Použité knižnice

Ku každému projektu napísanému v jazyku C/C++ sú automaticky prilinkované implicitné knižnice. Tie môžu dosahovať rádovo desiatky kilobytov a keďže sú z veľkej časti nevyužité, zbytočne zaberajú miesto z počiatočného priestoru, ktorý je k dispozícii. Preto

7 MinGW (*Minimalist GNU for Windows*) - sada kompilátorov.

jedným z parametrom, ktorý treba definovať je zákaz prilinkovania základných knižníc. Vo *Visual Studiu* nato slúži prepínač `Ignore All Default Libraries`, s parametrom **`/NODEFAULTLIB`**.

Jednou zo zakázaných knižníc je však *C-runtime* (*CRT*) knižnica, ktorá okrem iného zodpovedá za vstupnú funkciu do programu. Okrem spustenia konštruktorov globálnych premenných volá funkciu **`main()`**, príp. **`WinMain().`** Preto pri pokuse o preklad programu linker zahlási chybu o nenájdení vstupnej funkcie **`WinMainCRTStartup()`**. Riešením je dodefinovanie spomínanej funkcie, prípadne zadanie nového vstupného bodu. Ja som zvolila druhú možnosť a pomocou prepínača `Entry Point` som nastavila ako počiatočný bod vlastnú funkciu. Funkciu **`main()`** (**`WinMain()`**) teraz možno vynechať a definovaný nový vstupný bod využiť ako klasickú hlavnú funkciu.

Ďalšie funkcie, ktoré už nie sú k dispozícii kvôli vynechaniu implicitných knižníc, sú matematické funkcie. Vlastné implementácie použitých matematických funkcií sú podrobne popísané v nasledujúcej kapitole 4 Matematické funkcie.

Pri tvorbe výsledného programu sa takisto nedá zaobísť bez alokačných a dealokačných funkcií pre správu pamäte. Hoci nemožno použiť ich implementáciu poskytovanú *C-runtime* knižnicou, dá sa využiť ich obdoba poskytovaná rozhraním *WinAPI*[8] - **`GlobalAlloc()`** a **`GlobalFree()`**.

3.4 Nastavenia prepínačov

Vhodné nastavenie prepínačov je veľmi dôležité pre konečnú veľkosť programu a optimalizáciu kódu. Kompilátor prostredia *Visual C++* poskytuje veľké množstvo nastaviteľných parametrov. Každý z nich má určitý vplyv na výsledný program. Viac informácií o možných nastaveniach je dostupných na domovských stránkach *VS* [6].

Nasledujúci zoznam obsahuje vždy názov prepínača, hodnotu, ktorú mu definujeme a význam tohto nastavenia:

- `Optimization:` **`Minimize Size (/O1)`** – pri možnosti voľby prioritné vytvorenie malého kódu na úkor ostatných vlastností.
- `Favor Size or Speed Faver:` **`Small Code (/Os)`** – kompilátor minimalizuje veľkosť výsledného kódu na úkor rýchlosti.
- `Enable String Pooling:` **`Yes(/GF)`** – povolenie umiestnenia kópie identických reťazcov do výsledného súboru. Rovnaké reťazce sú teda uložené na jedinom mieste a tým sa stane výsledný program menší.

8 WinAPI (Windows API) - aplikačné rozhranie vyvinuté firmou *Microsoft* pre operačné systémy *Windows.*

- `Struct Member Alignment:` **`1 Byte (/Zp1)`** – zarovnanie jednotlivých položiek štruktúr na 1 byte.
- `Buffer Security Check:` **`No (/GS-)`** – zakázanie kontroly práce so zásobníkom. Pri precíznom programovaní by vypnutie nemalo mať žiadne negatívne následky.
- `Enable Run-Time Type Info:` **`No (/GR-)`** – v prípade povolenia prepínač pridá kód pre typovú kontrolu objektov behom programu. Pre potreby dema však kontrola nie je nevyhnutná a tak sa jeho vypnutím ušetrí ďalšie potrebné miesto.

Spomínané prepínače umožňujú zmenšiť výstupný súbor. Existuje však ešte niekoľko metód ako ušetriť ďalšie byty. Napríklad pri práci s číslami s plávajúcou desatinnou čiarkou používať namiesto dátového typu `double`[9] dátový typ `float`[10] (samozrejme tam, kde je presnosť typu `float` dostačujúca). Takisto používanie typu `char`[11] (`unsigned char`), prípadne jeho OpenGL obdoby `GLbyte` (`GLubyte`), pri práci, kde je rozsah -127 až +127 (0 - 255) prijateľný, ako napríklad rozsah jednotlivých farebných zložiek.

3.5 Kompilácia programov písaných v assembleri

Hoci je takmer celý program napísaný v jazyku C++, pripojenie zvukového doprovodu (kapitola 7.1 Hudobný podklad) si vyžaduje kompiláciu súboru za použitia assemblera[12]. Vhodnou voľbou sa ukázal *YASM* [8], nástupca rozšíreného assemblera *NASM*. Jeho integrácia do *Visual Studia* je veľmi jednoduchá a nevyžaduje žiadne špeciálne nastavenia.

9 `double` - 64 bitové číslo s plávajúcou desatinnou čiarkou, rozsah 3.4E +/- 38 (~15 číslic).

10 `float` - 32 bitové číslo s plávajúcou desatinnou čiarkou, rozsah 1.7E +/- 308 (~7 číslic).

11 `char`, `unsigned char`, `GLbyte`, `GLubyte` - 8 bitové celé číslo.

12 assembler - program pre preklad jazyka symbolických inštrukcií do strojového kódu.

4 Matematické funkcie

Základné matematické funkcie sa po zákaze prilinkovania implicitných knižníc stali neprístupné. Preto je ich nutné dodatočne naimplementovať, prípadne využiť obdobu poskytovanú matematickou knižnicou. Vhodným spôsobom pre ich definovanie je využitie assembleru a priamych inštrukcií *FPU*[13]. Podrobný popis možných inštrukcií je dostupný v [8].

Ako jednoduchý príklad možno uviesť výpočet absolútnej hodnoty (**math_abs()**):

```
float math_abs( const float x) {
        float r;
        _asm fld  dword ptr [x];     //uloženie hodnoty x na vrch zásobníku
        _asm fabs;                   //absolútna hodnota
        _asm fstp dword ptr [r];     //načítanie hodnoty z vrchu zásobníka do
                                     //premennej r
        return r;
}
```

Obdobne sú implementované funkcie **math_sinf()** a **math_cosf()**, ktoré využívajú inštrukcie **fsin** a **fcos** či odmocnina desatinného čísla **math_sqrt()** (inštrukcia **fsqrt**). O niečo zložitejšou funkciou je umocňovanie desatinného čísla na neceločíselnú konštantu: **math_powf()** [20]:

```
float math_powf(const float x, const float y) {
        _asm fld     dword ptr [y]   //uloženie hodnoty y na vrch zásobníku
        _asm fld     dword ptr [x]   //uloženie hodnoty x na vrch zásobníku
        _asm fyl2x                   //y*log2x
        _asm fld1                    //uloženie hodnoty 1 na vrch zásobníka
        _asm fld     st(1)           //uloženie hodnoty z st(1) vrch zásobníku
        _asm fprem                   //časť čísla za desatinnou čiarkou
        _asm f2xm1                   //2^x-1
        _asm faddp   st(1), st(0)    //st(1) + st(2)
        _asm fscale                  //hodnotu z st(1) zaokrúhli nadol a pripočíta
                                     //k exponentu st(0)
        _asm fxch                    //prehodenie st(1) a st(2)
        _asm fstp    st(0)           //načítanie hodnoty z vrchu zásobníka do st(0)
        _asm fstp    dword ptr [res];  //načítanie hodnoty z vrchu zásobníka do
                                       //premennej res
```

13 FPU (*Floating Point Unit*) – koprocesor pre operácie s číslami s plávajúcou desatinnou čiarkou.

```
        return res;
    }
```

Pomocou inštrukcií FPU je implementovaná aj funkcia pre výpočet dolnej celej časti desatinných čísel **math_ifloorf()** a zvyšku po delení desatinného čísla iným desatinným číslom **math_fmodf()**[].

Funkcie pre výber minima a maxima z dvoch desatinných čísel **math_min()** a **math_max()** sú definované podmienkou:

```
#define math_min(a, b)      (a < b) ? a:b
#define math_max(a, b)      (a > b) ? a:b
```

Ďalej sú vytvorené funkcie pre :

- lineárnu interpoláciu **math_linearInterpolate()**:

```
float math_linearInterpolate(float a, float b, float t) {
        return a*(1.0f - t) + b*t;
    }
```

- kosínusovú interpoláciu **math_cosInterpolate()**:

```
float math_cosInterpolate(float a, float b, float t) {
        float value = (1.0f - math_cosf(t*PI))*0.5f;
        return math_linearInterpolate(a, b, value);
    }
```

- získanie hodnoty Beziérovej krivky tretieho stupňa v priestore určenej riadiacimi bodmi pre zadaný parameter, **math_bezierValue()**:

```
Coords math_bezierValue(BezierSegment *input, float t) {
        Coords ret;
        ret.x = ret.y = ret.z = 0.0f;
        float bernstein[4] = {1.0f, 3.0f, 3.0f, 1.0f};//hodnoty Bernsteinovych pol.
        Coords tmp;
        //B0*t^0*(1-t)^3 + B1*t^1*(1-t)^2 + B2*t^2*(1-t)^1 + B3*t^3*(1-t)^0
        for (int i=0; i<4; i++) {
                tmp = input->points[i];
                tmp *= (bernstein[i]*math_powf(t, float(i))*math_powf((1.0f - t),
                        3.0f - float(i)));
                ret += tmp;
        }
        return ret;
    }
```

- orezanie hodnoty do intervalu `<0.0, 1.0>` **`math_clamp01()`**.
- násobenie vektora s maticou **`math_vectorByMatrix()`**.
- funkcia *random*, ktorá vráti náhodnú hodnotu (je podrobne popísaná v nasledujúcej podkapitole).

4.1 Funkcia random

Veľa algoritmov použitých vo výslednom intre je založených na pseudonáhodnom generátore náhodných čísel, ktorý vracia náhodné hodnoty na základe vstupnej konštanty. Pri implementácii som vychádzala z algoritmu, ktorý popísal *Iñigo Quilez* [20], člen demoskupiny *RGBA*. Viac o tomto algoritme je možné nájsť na ich internetových stránkach [19]. Náhodné číslo odvodené z inicializačnej hodnoty (`seed`) sa získa pomocou bitových operácií. Presnejšie, pre generovanie čísla s plávajúcou desatinnou čiarkou z intervalu `<0.0, 1.0>`, je definovaná funkcia

```
float math_randf_01() {
	seed *= 16807;
	unsigned int ret = (seed & 0x007fffff) | 0x3f800000;
	return( *((float*)&ret) - 1.0f );
}
```

a pre celé čísla

```
int math_randi() {
	seed = seed * 0x343FD + 0x269EC3;
	return ((seed>>16) & 32767);
}
```

Pre prehľadnejšie používanie boli dodefinované funkcie, ktoré generujú:

- desatinné číslo z ľubovoľného intervalu `<a, b>`
- celé číslo z intervalu `<a, b>`
- pole náhodných desatinných čísel z definovaného intervalu `<a, b>` (využívané pri inicializácii Perlinovho šumu).

5 Použité techniky

Obsahom intra je život kvetiny, a to od počiatočného semienka, cez delenie buniek až po rast kvetiny. Z toho dôvodu bolo potrebné riešiť problematiku generovania trojrozmerných objektov. Keďže je rastlina zasadená na otvoreného prostredia, bolo treba implementovať vykresľovanie zeme a oblohy. V intre sú ďalej modelované svetelné lúče, vietor, búrka a pod. Na konci intra sú krátke titulky, a preto je jedna z nasledujúcich podkapitol venovaná tvorbe fontov. Vo zvyšných častiach sú popísané potrebné modelovacie a textúrovacie techniky, ako aj rôzne efekty pre zvýšenie grafickej hodnoty výsledného diela. Všetky metódy majú spoločný rys – z minimálneho (prípadne nulového) počtu vstupných dát vygenerovať hodnoty pre komplexný popis objektu s požadovanými vlastnosťami. Súčasne musí byť splnená dôležitá podmienka, aby všetky výpočty prebiehali v reálnom čase. Zo spomínaných efektov je to napríklad použitie objektu framebufferu pre vytvorenie tieňa alebo vykresľovanie scény len do určitej oblasti okna použitím testu šablónou.

5.1 Textúry

V dôsledku obmedzenia výsledného programu na 64 kB nie je možné textúry načítať z vopred uložených obrázkových dát. Z toho dôvodu je potrebné si textúry vygenerovať na začiatku, prípadne počas vykonávania programu. Takéto textúry sa nazývajú procedurálne a patrí medzi ne napríklad tehlová stena, tráva, oheň alebo mraky. Spoločným znakom je určitá pravidelnosť vo vzore, ktorá môže byť matematicky popísaná. Na tvorbu procedurálnych textúr sa používajú techniky ako Perlinov šum, zakrivenie, turbulencia, fraktály, voronoiov diagram, rôzne matematické funkcie alebo zmeny vo farebných kanáloch.

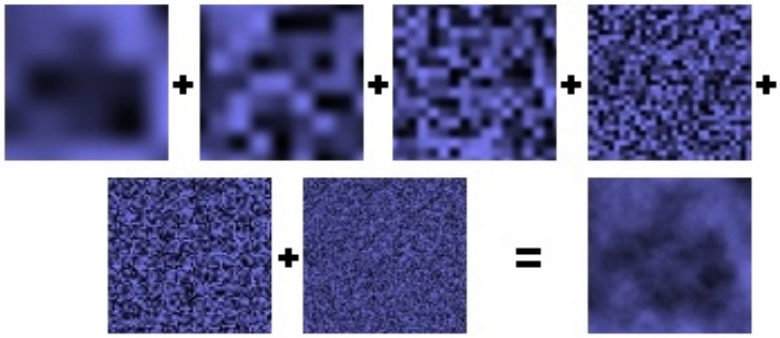
V tejto práci boli využité nasledujúce techniky – Perlinov šum, celulárne textúry a pre vytvorenie jemného reliéfu metóda známa ako *bump mapping*. Tieto techniky sú podrobnejšie rozpísané v nasledujúcich podkapitolách.

5.1.1 Perlinov šum

Perlinov šum je metóda pre generovanie šumovej funkcie, ktorú predstavil *Ken Perlin*[14] už v roku 1985. Dnes patrí medzi najpoužívanejšie techniky na tvorbu procedurálnych textúr.

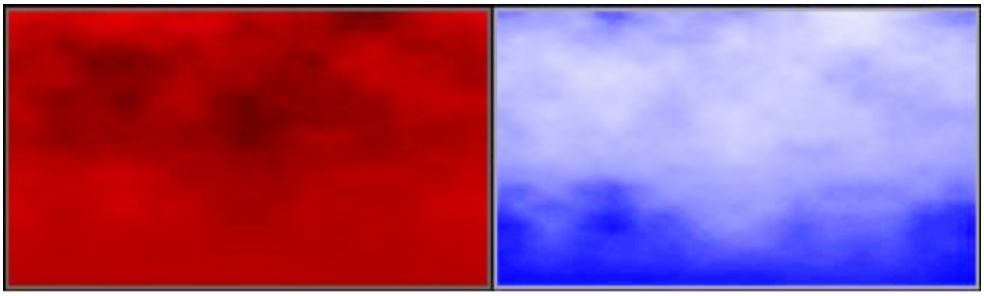
14 *Ken Perlin* - profesor na Univerzite v New Yorku, držiteľ mnohých ocenení v oblasti počítačovej grafiky [11].

Metóda je založená na sčítavaní tej istej šumovej funkcie s rôznou intenzitou a škálou (Obrázok 1). Ako základný šum sa využíva pseudonáhodný generátor, ktorý vracia náhodne zvolené číslo, avšak pre ten istý vstup vráti vždy rovnaký výsledok. Každá vrstva je súčasne vyhladená zvolenou interpolačnou technikou [2]. Najjednoduchšia je lineárna interpolácia, ktorá však často nedosahuje požadovanú kvalitu. Kompromisom medzi rýchlosťou a kvalitou je kosínusová interpolácia, ktorú som zvolila aj ja. Výsledná textúra môže byť jedno- až štvorrozmerná. 4D textúry, nazývané aj hypertextúry, sa využívajú na tvorbu 3D objektov so zmenou v čase, ako napríklad oheň alebo dym. Najväčšie uplatnenie našiel Perlinov šum pri tvorbe textúr ako sú mramor alebo drevo.



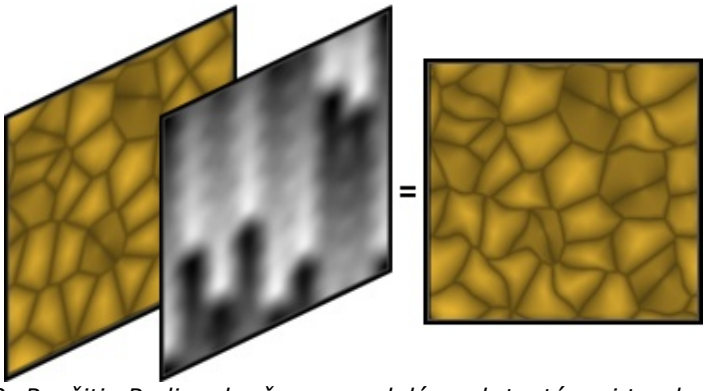
Obrázok 1: Šum s rôznou frekvenciou a výsledné sčítanie [10].

V projekte je Perlinov šum použitý na tvorbu mrakov.

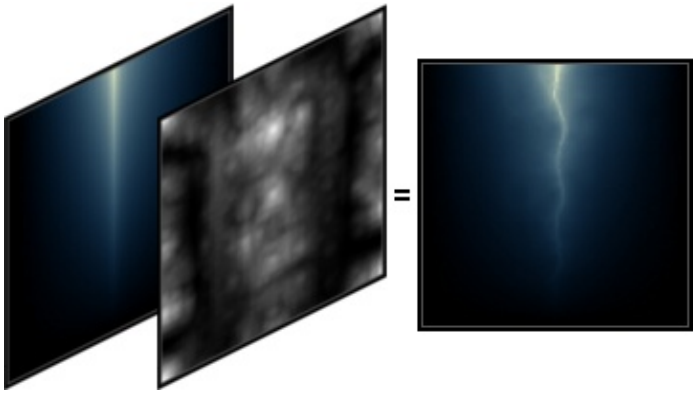


Obrázok 2: Perlinov šum použitý na tvorbu oblohy s mrakmi.

Ďalej je použitý na pridanie určitej nepravidelnosti zeme vytvorenej pomocou celulárnych textúr (Obrázok 3) a pri tvorbe blesku (Obrázok 4).



Obrázok 3: Použitie Perlinovho šumu a celulárnych textúr pri tvorbe zeme.



Obrázok 4: Použitie Perlinovho šumu pri tvorbe blesku.

Dvojrozmerný Perlinov šum sa dá využiť ako výšková mapa pre generovanie terénu. Index do poľa hodnôt určuje súradnice v dvoch smeroch a hodnota v tomto bode definuje ostávajúcu súradnicu trojrozmerného priestoru. Trojrozmerný šum je použitý pri modelovaní vetra, kde dva rozmery udávajú výškovú mapu (teda akési „zvlnenie“ vetra) a tretí rozmer je použitý pre zmenu v čase. Podrobnejšie je technika tvorby vetra a terénu popísaná v kapitole 5.7 Výšková mapa.

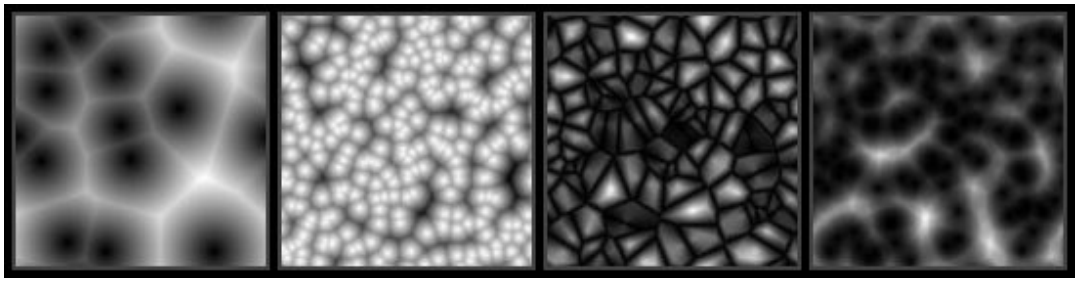
5.1.2 Celulárne textúry

Celulárne textúry ponúkajú výborný nástroj k tvorbe procedurálnych textúr najmä vďaka veľkej rozmanitosti, pričom v algoritme dochádza k minimálnym zmenám. Algoritmus je založený na Voronoiovom diagrame, teda diagrame, ktorý vychádza z prerozdelenia oblasti na základe vzdialeností od vopred určených bodov [2][12]. Výsledná textúra potom závisí od zvolenej metriky.

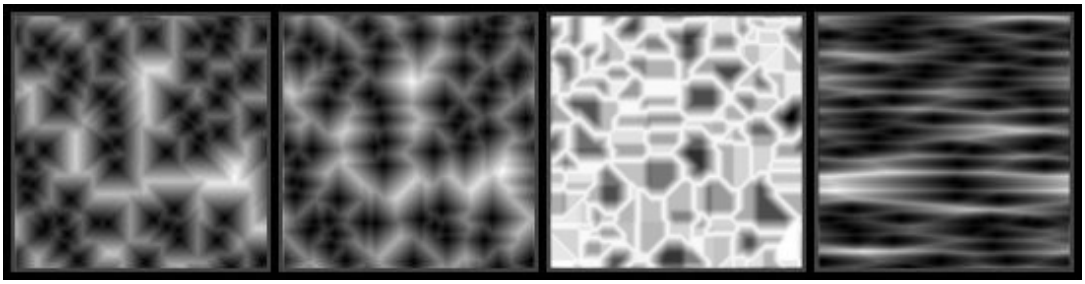
Algoritmus prebieha v nasledujúcich krokoch:

- `vygeneruj` **`n`** `náhodných bodov`
- `pre každý bod textúry zisti vzdialenosť od najbližšieho bodu`
- `každému bodu textúry priraď farbu podľa jeho normalizovanej vzdialenosti`

Pri veľkom množstve vstupných bodov je zistenie najbližšieho bodu pre každý pixel časovo veľmi náročnou operáciou. Možným riešením je obmedzenie počiatočných bodov alebo generovanie bodov postupne do pravidelnej mriežky. V tom prípade by sa vždy vyberal bod maximálne z deviatich možných hodnôt.



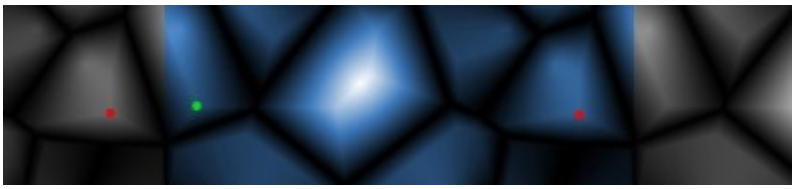
*Obrázok 5: Celulárne textúry s použitím rôznych metrík. Zľava doprava (dist1 (dist2) – Euklidovská vzdialenosť (druhého) najbližšieho bodu): dist1, 1 – dist1, dist2 - dist1, dist1*dist2* [12].



*Obrázok 6: Celulárne textúry s použitím rôznych metrík. Zľava doprava (dist1.x (dist1.y) – vzdialenosť najbližšieho bodu v smere osi x (y), podobne pre dist2) : Manhattanská vzd., Chebyshevova vzd., (dist1.x + dist1.y) - (dist2.x + dist2.y), dist1.x*dist1.x – dist1.y.*

Pre vytvorenie textúry, ktorú je možné opakovať v oboch smeroch, je nutné zaistiť napojenie na všetkých štyroch hranách. Preto treba upraviť výpočet vzdialeností tak, akoby boli jednotlivé štvorce poskladané vedľa seba a bod textúry ovplyvňovali aj body ležiace vo vedľajšom štvorci (Obrázok 7). To je možné dosiahnuť nasledujúcou zmenou v algoritme:

- definujme **dx**(**dy**) ako rozdiel v x-ovom(y-ovom) smere a **text_width**(**text_height**) ako šírku(výšku) textúry
- ak je **dx > text_width/2** potom **dx = text_width – dx**
- ak je **dy > text_height/2** potom **dy = text_ height – dy**



Obrázok 7: Pri počítaní vzdialenosti pre zelený bod vzhľadom na červený, bude podmienka splnená a uplatní sa zmena.

V práci bola použitá celulárna textúra na generovanie suchej a popraskanej zeme.

5.1.3 Bump mapping

Bump[15] *mapping* je technika, pomocou ktorej je možné vytvoriť dojem reliéfu bez zmeny geometrie objektu. Využívajú sa čiernobiele alebo normálové textúry. Pomocou hodnôt v textúre sú spočítané vychýlenia normál pre každý bod. Vďaka tomu sú vypočítané nové hodnoty odrazu svetla, čo je možné využiť pre navodenie trojrozmernému dojmu. Bump mapping je vhodný pri miernych zmenách v povrchu, ako je napríklad hrboľatá zem, kôra pomaranča alebo tkanina. Pre väčšie zmeny, ako je zvlnenie terénu, je nutné použiť iné techniky [2].

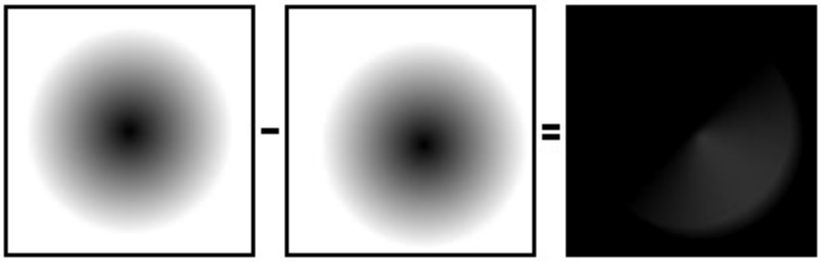
Obrázok 8: Ukážka bump mappingu [13].

15 bump (angl.) - nerovnosť, výmoľ,...

5.1.3.1 Emboss bump mapping

Podľa zvolenej techniky výpočtu a charakteristike vstupných dát sa rozlišuje niekoľko rôznych podkategórií bump mappingu. Medzi ne patrí aj tzv. *emboss*[16] *bump mapping*, ktorý je implementovaný vo výslednom programe. Emboss BM je jedna z najjednoduchších metód bump mappingu. Na rozdiel od pôvodnej myšlienky, kde je osvetlenie vypočítané pre každý pixel na základe jeho normály, emboss BM sa líši od ostatných metód tým, že nepočíta tieňovanie priamo na základe odrazu svetla a normál. Namiesto toho využíva posun dvoch výškových máp [14].

Prvá varianta emboss BM je založená na posunutí dvoch identických máp v závislosti od polohy svetelného zdroja a ich vzájomnom odčítaní. Vďaka tomu sa vytvoria svetlé a tmavé oblasti. Následným aplikovaním farebnej textúry sa dosiahne požadovaného efektu. Tento postup je veľmi priamočiary a ľahko implementovateľný. Na druhej strane trpí viacerými nedostatkami ako je napríklad jeho viazanosť na takmer rovné povrchy či vznik nežiaducich artefaktov (Obrázok 9).



Obrázok 9: Klasický emboss bump mapping.

Druhá metóda spočíva v počiatočnom vygenerovaní dvoch nových výškových máp zo vstupnej textúry. Prvá má polovičnú intenzitu a druhá invertovanú polovičnú intenzitu. Novovzniknuté textúry sa vzájomne posunú v smere osvetlenia a v poslednej fáze sa skombinujú s pôvodnou textúrou. Táto metóda dosahuje o čosi lepšie výsledky ako vyššie spomínaný prístup. Stále je však použiteľná len na relatívne rovné povrchy (Obrázok 10).

Obrázok 10: Emboss bump mapping s použitím upravených výškových máp.

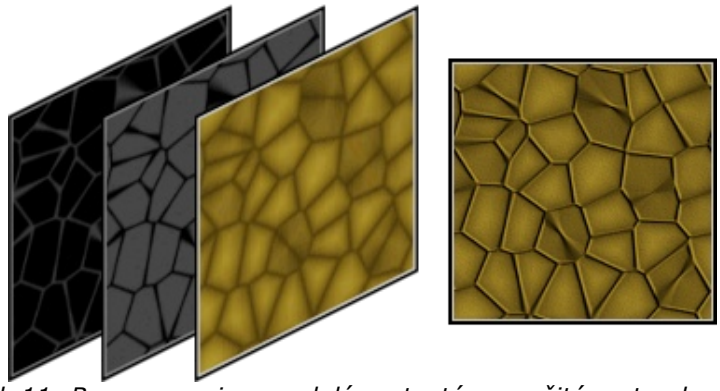
16 emboss (angl.) - vytlačiť, vyrezať,...

Keďže je vo výslednom programe bump mapping použitý pri zobrazení rozpraskanej zeme, ktorá sa dá považovať za viacmenej rovný podvrh, emboss bump mapping s použitím vygenerovaných výškových máp je vhodná metóda. Okrem dostačujúcich vizuálnych výsledkov nie je časovo príliš náročná a skladanie textúr je pomocou OpenGL ľahko implementovateľné.

5.1.3.2 Aplikácia vo výslednom programe.

Výsledná mump mapa je vytvorená prahovaním a zašumením textúry zeme vytvorenej pomocou celulárnej techniky. Vďaka zašumenie sa dosiahne jemne hrboľatého efektu a zem nepôsobí hladko.

Plocha, na ktorú sa nanášajú textúry, leží v rovine určenej rovnicou `y=0`. To znamená, že normála je vždy určená vektorom `(0, 1, 0)`. Vzájomný posun textúr v smere osí `x` a `z` sa určí v závislosti od polohy svetelného zdroja, skúmaného vrcholu a faktoru, pomocou ktorého môže byť posun znásobený, prípadne utlmený. Pri samotnom nanášaní textúr sa využívajú funkcie OpenGL pre multitextúrovanie, ktoré priradia textúrovacie súradnice všetkým trom použitým textúram. Operácie skladanie textúr sú implementované vo fragment shaderi (viď. Kapitola 6.2.2 Bump mapping).



Obrázok 11: Bump mapping a celulárne textúry použité na tvorbu zeme.

5.2 3D objekty

Keďže priestor, ktorý je k dispozícii, neponúka dostatočné možnosti pre uchovanie všetkých vrcholov zložitých 3D modelov, je treba zvoliť metódu, ktorou môžu byť objekty vymodelované z menšieho množstva základných riadiacich bodov. Výsledný objekt sa získa vygenerovaním nových bodov na základe daného kritéria. V tejto práci bol použitý algoritmus prerozdelenia známy ako *Catmull-Clark subdivision*. Pomocou tejto metódy sú vymodelované všetky použité objekty. Algoritmus pracuje s triedou `Mesh`, o ktorej pojednáva nasledujúca podkapitola.

5.2.1 Štruktúry a triedy pre popis 3D objektov

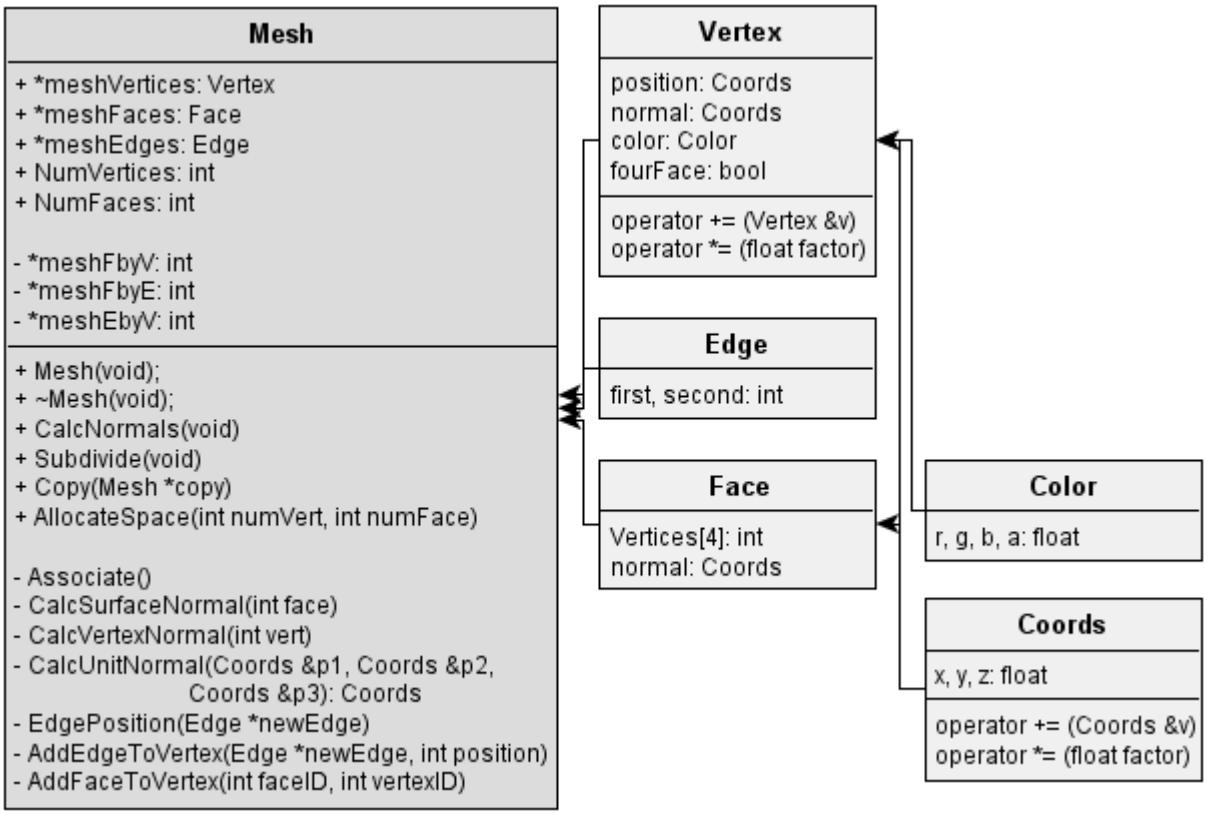
Keďže sa pracuje v trojrozmernom priestore, bola pre uloženie súradníc vytvorená štruktúra `Coord`, ktorá obsahuje tri premenné typu `float` reprezentujúce tri súradnice. Pre prácu s farbami bola implementovaná štruktúra `Color`, ktorá má štyri premenné typu `float` reprezentujúce štyri zložky farby v móde *RGBA*[17].

Najjednoduchším geometrickým primitívom v priestore je bod. V programe je bod reprezentovaný štruktúrou `Vertex`. Pre správny výpočet tieňovania a farby musí byť v OpenGL pre každý vrchol definovaná jeho normála a farba. Pre potreby prerozdelenia objektu, ktorý je popísaný v nasledujúcej podkapitole, je pri každom vrchole navyše uložená informácia, či bod inciduje s tromi alebo štyrmi stenami (resp. hranami). Spojením dvoch bodov vznikne hrana. Štruktúra `Edge` obsahuje dva indexy do poľa vrcholov, ktoré určujú krajné body hrany.

Pre kompletný popis objektu treba okrem vrcholov a hrán, definovať aj jeho steny. Za týmto účelom bola vytvorená štruktúra `Face`. Algoritmus prerozdelenia je prispôsobený práci so stenami tvorenými štyrmi vrcholmi. Preto štruktúra `Face` obsahuje štyri premenné typu `int`, ktoré slúžia ako indexy do poľa vrcholov. Navyše obsahuje informáciu o hodnote normály danej steny.

Triedou, ktorá zastrešuje všetky hore uvedené štruktúry, je trieda `Mesh`. Tá obsahuje polia vrcholov, hrán a stien objektu, vzájomné prepojenia primitív a metódy potrebné na prerozdelenie objektu, príp. skopírovanie jedného objektu do iného. Kompletný obsah triedy `Mesh`, hore uvedených štruktúr a ich vzájomných vzťahov znázorňuje Obrázok 12.

17 RGBA (Red Green Blue Alpha) - farba určená hodnotami červenej, zelenej a modrej zložky. Hodnota štvrtej zložky definuje priehľadnosť.

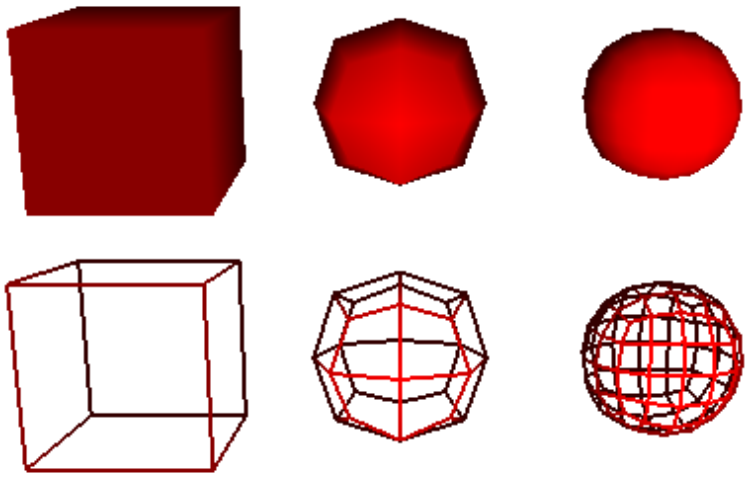


Obrázok 12: Znázornenie použitých tried a štruktúr pre popis trojrozmerných objektov.

5.2.2 Prerozdelenie algoritmom Catmull-Clark (Catmull-Clark subdivision)

Algoritmus Catmull-Clark vychádza z rekurzívneho prerozdeľovania plôch trojrozmerného telesa za účelom vytvárania hladkých povrchov. Prvý krát bol použitý v roku 1978, kedy *Edwin Catmull* a *Jim Clark* použili tento algoritmus pre zovšeobecnenie použitia bikubických uniformných B-splinových plôch z pravouhlých objektov na objekty s ľubovoľnou topológiou [9].

Pri tvorbe trojrozmerných objektov potom stačí menší počet hlavných riadiacich bodov a postupným zjemňovaním stien objektu je možné dosiahnuť ľubovoľne hladký povrch. Vďaka tomu napríklad na vymodelovanie gule stačí osem hlavných určujúcich bodov, ktorými sa zadá prvotná kocka. Ostatné body sú následné dopočítané a posunuté v závislosti od polohy susediacich bodov (Obrázok 13).



Obrázok 13: Vymodelovanie gule pomocou zjemňovania kocky algoritmom Catmull-Clark.

Algoritmus prebieha v nasledujúcich krokoch:

- pre každú stenu vypočítaj nový bod (**face point**) ako priemer všetkých bodov steny
- pre každú hranu vypočítaj nový bod (**edge point**) ako priemer dvoch pôvodných bodov hrany a dvoch **face point** patriacich priľahlým stenám
- pridaj objektu nové hrany spájajúce novovzniknuté **face** a **edge point**
- pre každý pôvodný bod **P** spočítaj priemer všetkých **face point** patriacich stenám incidentným s bodom **P** (ozn. **F**) a priemer všetkých stredov hrán incidentných s bodom **P** (ozn. **R**)
- posuň každý pôvodný bod **P** do bodu daného vzťahom
- **(F + 2*R + (n-3)*P)/n,** kde **n** je počet incidentných stien (resp. hrán)

Obrázok 14: Vymodelovanie kvetiny pomocou postupného zjemňovania.

V projekte bol vybraný štvoruholník ako základné primitívum na prerozdelenie, pričom bod inciduje vždy s tromi alebo štyrmi stenami (resp. hranami). Tomu bol prispôsobený aj použitý algoritmus. Po každom prerozdelení je nutné prepočítať normály vo všetkých bodoch pre korektné tieňovanie a vykresľovanie.

5.3 Tiene

Pre zvýšenie reálnosti scény boli pridané tiene. V OpenGL existuje niekoľko rôznych spôsobov ako vytvoriť tiene, napríklad pomocou stencil testu a tieňového telesa. V práci bol zvolený postup založený na algoritme publikovanom na stránkach [15], ktorý využíva *framebuffer object* (ďalej FBO) a vstavanú funkciu GLSL **`shadow2DProj()`** [3].

Pri inicializácii je potrebné vytvoriť FBO a textúru pre uloženie tieňovej mapy. Tej je nutné nastaviť požadované hodnoty pre opakovanie, interpoláciu a vlastnosti potrebné pre jej použitie ako hĺbkovej mapy. Medzi ne patrí napríklad metóda porovnávania hodnôt, porovnávacia funkcia a spôsob, akým sa má hodnota v textúre meniť:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_R_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE, GL_INTENSITY);
```

Pri samotnom renderovaní scény je nutné:

- Naplniť FBO:

```
// zapnúť zápis do FBO
glBindFramebufferMD(GL_FRAMEBUFFER, shadow_FBO);

// v prípade, že FBO má rôzne rozlíšenie, treba upraviť veľkosť viewportu. Väčšie
// rozlíšenie umožňuje hladšie okraje tieňov, avšak na úkor časovej náročnosti
glViewport(0, 0, width * SHADOW_STEP, height * SHADOW_STEP);
// vynulovať hĺbkový buffer
glClear(GL_DEPTH_BUFFER_BIT);

// vypnúť zápis farebných informácii – dôležitá je len vzdialenosť od kamery
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// nastaviť projekciu z bodu, v ktorom sa nachádza svetlo smerom určeným smerom
// svetla
glMatrixMode(GL_PROJECTION);
```

```
glLoadIdentity();
gluPerspective(45, (float)width/float(height), 1.0, 100);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(light_pos[0], light_pos[1], light_pos[2], light_dir[0], light_dir[1],
          light_dir[2], 0, 1, 0);

// nastaviť orezávanie privrátených stien (aby sa zabránilo samozatieňeniu)
glCullFace(GL_FRONT);

// vykresliť objekty, na ktoré sa majú premietnuť tiene a ktoré tieň vrhajú
...
```

- Pozíciu bodu treba správne premietnuť do súradníc textúry. Výsledná matica sa získa zložením modelovo-pohľadovej matice (ktorá transformuje modelovacie súradnice bodov do svetových súradníc a následne do súradnicového systému s počiatkom v mieste svetelného zdroja), projektívnej matice (definuje viditeľný objem, orezávacie roviny a pomer strán) a matice, ktorá transformuje hodnoty získané predchádzajúcimi transformáciami z rozsahu `<-1, 1>` do intervalu `<0, 1>`. Výsledné hodnoty potom môžu byť priamo použité ako indexy pre prístup do tieňovej mapy.
- Po vytvorení treba vypnúť renderovanie do FBO, nastaviť viewport a transformačné matice do pôvodných hodnôt a vypnúť orezávanie privrátených stien.
- Pri následnom renderovaní scény sa tiene pripájajú pomocou vertex a fragment shaderu, ktorý je bližšie popísaný v kapitole 6.2.1 Tiene.

5.4 Point sprity

Od verzie OpenGL 2.0 sú súčasťou knižnice aj bodové sprity (angl. *point sprites*) s textúrou, ktorej súradnice sú generované v závislosti od definovaných bodov. Tie umožňujú lepšiu kontrolu nad spracovaním veľkých bodov. Bodové sprity určujú spôsob, ako sú generované dáta fragmentov rozšíreného bodu. Pri inicializácii je možné pomocou funkcie **`glPointParameter{if}v()`** nastaviť parametre, ktoré určujú orientáciu textúry, maximálnu, minimálnu a základnú veľkosť bodov či koeficienty kvadratickej rovnice, ktorá určuje postupné zmenšovanie bodov v závislosti od vzdialenosti od kamery. Pri samotnom použití je potrebné:

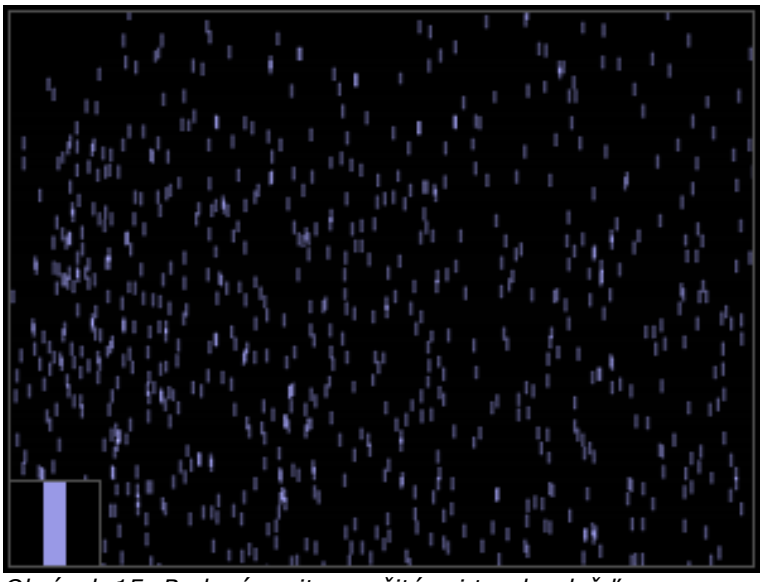
```
// povoliť textúrovanie a bodové sprity
glEnable(GL_TEXTURE_2D);
glEnable(GL_POINT_SPRITE);

// nastaviť veľkosť bodov
glPointSize(8.0f);

// textúrovacej jednotke povedať, že sa jedná o bodové sprity
glTexEnvi(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);

// vykresliť body
glVertex3f(x, y, z);
```

Bodové sprity som použila pri animácii dažďových kvapiek, kde každá kvapka je jeden otextúrovaný bod (Obrázok 15).



Obrázok 15: Bodové sprity použité pri tvorbe dažďa.

5.5 Stencil test

Pred zápisom fragmentu do výstupného bufferu podstúpi každý fragment niekoľko testov. Na základne týchto testov môže byť fragment vylúčený z ďalšieho spracovania. Medzi spomínané testy patrí napríklad depth test[18], alpha test[19], scissor test[20] a takisto stencil test. Stencil test, inak nazývaný aj test šablónou, porovnáva hodnotu prichádzajúceho fragmentu s hodnotou uloženou v pamäti šablóny. Na základe definovanej funkcie sa rozhodne, či fragment testom prešiel a ako sa má zmeniť hodnota uložená v pamäti v závislosti od výsledku porovnania.

V práci bol použitý jednoduchý stencil test pre zobrazenie „pohľadu ďalekohľadom“ (Obrázok 16). Pri inicializácii intra sa naplnení pamäť stencil bufferu. V priebehu programu sa zapne stencil test a vykresľuje sa len do určenej časti framebufferu. Vytvorenie stencil bufferu prebieha v nasledujúcich krokoch:

- vytvorenie masky (naplnenie pamäti)

```
// povolenie stencil testu
glEnable(GL_STENCIL_TEST);

// nastavenie ortogonálnej (kolmej) projekcie
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-width/height, width/height, -1.0, 1.0);

//nastavenie modelview matice
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

 // nastavenie hodnoty ktorá sa nastaví pri následnom premazaní bufferu
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);
```

18 *depth test* (test hĺbky) - test na určenie vzájomnej polohy a prekrytia objektov a z toho vyplývajúcej viditeľnosti .

19 *aplha test* (test hodnoty alpha) - porovnávanie štvrtej zložky farby s prahovou hodnotou, kde porovnávacia funkcia je zadaná užívateľom.

20 *scissor test* („nožnicový“ test) - test polohy fragmentu vzhľadom na danú obdĺžnikovú oblasť výstupného bufferu.

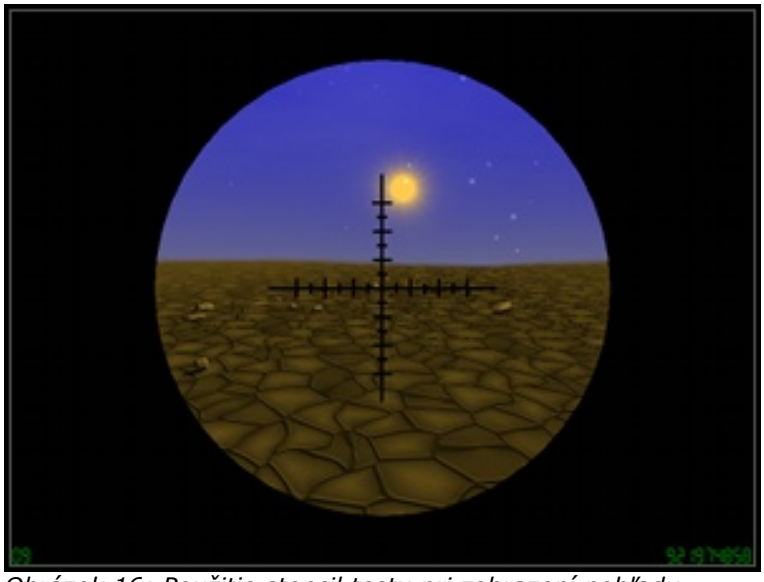
```
// nastavenie porovnávacej funkcie a operácií vykonaných v pamäti pri rôznych
// výsledkoch testu
glStencilFunc(GL_ALWAYS, 0x1, 0x1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

// vykreslenie vnútorného kruhu
...

// zmena operácií vykonávaných na pamäti
glStencilOp(GL_ZERO, GL_ZERO, GL_ZERO);

// vykreslenie čiar do vnútorného kruhu
...

// zakázanie stencil testu
glDisable(GL_STENCIL_TEST);
```



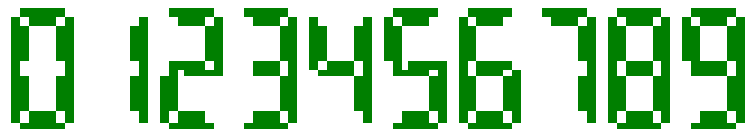
Obrázok 16: Použitie stencil testu pri zobrazení pohľadu ďalekohľadom.

5.6 Fonty

OpenGL nepodporuje vykresľovanie znakových reťazcov priamo. Namiesto toho umožňuje rasterizáciu tzv. bitových máp[21], pomocou ktorých je možné jednotlivé znaky vykresliť.

21 bitová mapa - v tomto prípade myslená ako pravouhlé pole núl a jednotiek.

Bitová mapa musí byť zarovnaná na osem bitov, aj keď samotná oblasť vykresľovanej bitmapy môže byť ľubovoľná. Pomocou bitových máp sú v intre vytvorené digitálne číslice (Obrázok 17).



Obrázok 17: Digitálne číslice vytvorené pomocou bitmapy.

Aby programátor nemusel každému znaku definovať vlastnú bitmapu, ktorú by v prípade úpravy musel prepočítavať, je možné použiť *display listy*. V grafickom kontexte aplikácie sa vytvorí písmo s požadovanými vlastnosťami (veľkosť, aliasing, typ písma, ...) a vygeneruje sa príslušný počet display listov [17]:

```
// vytvorenie fontu
SelectObject((HDC)hdc, CreateFont(... );

// vytvorenie príslušného počtu display listov
font_alphabet = glGenLists(96);

// naplnenie display listov (od 32. znaku ak je použitá ANSI množina znakov)
wglUseFontBitmaps((HDC)hdc, 32, 96, font_alphabet);
```

Reťazec `text` sa potom vykreslí volaním príslušných display listov na danej pozícii `[x, y]`.

```
glWindowPos2iMD(x, y);
glPushAttrib(GL_LIST_BIT);
glListBase(font_alphabet - 32);
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);
glPopAttrib();
```

5.7 Výšková mapa

Pre generovanie zvlnených terénov existuje v počítačovej grafike niekoľko rozličných metód. Medzi ne je možné zaradiť napríklad:

- *Midpoint displacement* – rekurzívna metóda, pri ktorej sa vypočíta stredový bod štvorca ako priemerná hodnota jeho vrcholov a následne sa vrchol posunie o náhodnú hodnotu. Tým sa pomocou pôvodného štvorca vygenerujú štyri nové. Iteratívnym delením pokračujeme až k požadovanému rozlíšeniu.
- *Fault formation* (metóda zlomu) – takisto rekurzívna metóda, pri ktorej sa jednej polovici bodov zlomu pripočíta náhodná hodnota.
- *Height map* (výšková mapa) – generovanie rastrových dát. Hodnota v každej bunke rastru vyjadruje výšku bodu na daných súradniciach v rovine určených pozíciou v rastri. Táto metóda bola použitá pre modelovanie terénu a vetra vo výslednom programe.

Pre vytvorenie výškovej mapy v tejto práci bol použitý Perlinov šum (kapitola 5.1.1 Perlinov šum). Mapa sa vykresľuje pomocou tzv. *triangle stripov*. Ide o pás tvorený trojuholníkmi, pričom každý ďalší bod definuje nový trojuholník s vrcholmi určenými predchádzajúcimi dvomi bodmi a novozadaným bodom. Súradnica v smere osi `y` na pozícii `[x, z]` je daná hodnotou výškovej mapy s indexom `[x][y]`.

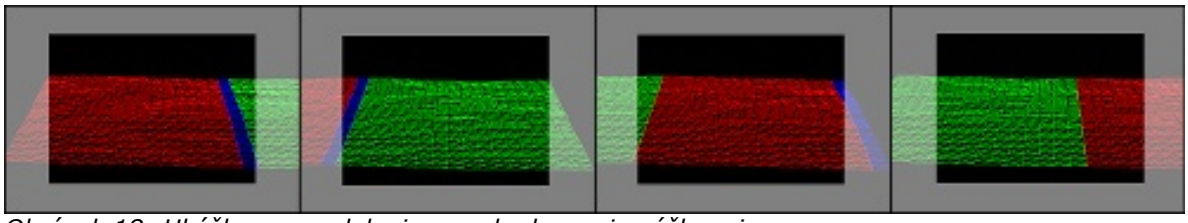
5.7.1 Vietor

Modelovanie vetra je pomerne náročnou úlohou. V skutočnom svete vietor znamená jednoduché prúdenie vetra a nedá sa pozorovať. Dajú sa však zaznamenať javy, ktoré ho sprevádzajú, ako vlnenie trávy, pohyb listov či jemné častice, ktoré vietor unáša. Keďže sa vo výslednom deme tráva ani stromy nevyskytujú, vietor je znázornený prúdením jemného prachu. Na jeho vykreslenie by bolo možné použiť častícový systém. Avšak pri tak veľkom množstve častíc, ktoré by bolo potrebné vykresliť, by výpočtová náročnosť rýchlo narastala. Finálnou technikou je dynamická výšková mapa vytvorená pomocou trojrozmerného Perlinovho šumu.

Šum znázornený v priestore, kde indexy `[x][y][z]` do vygenerovaného poľa hodnôt určujú jednotlivé súradnice, dosahuje na rezoch rovnobežných s rovinami `x=0`, `y=0` a `z=0` rovnaké charakteristiky ako dvojrozmerný Perlinov šum. Preto je možné šum v smere jednej zo súradnicových osí rozdeliť na jednotlivé vrstvy, z ktorých každá sa dá použiť ako výšková mapa. Vďaka trojrozmernému charakteru Perlinovho šumu sa pritom jednotlivé susediace vrstvy od seba líšia len minimálne (podobne ako dva susedné body v dvojrozmernom šume). Postupným vykresľovaním jednotlivých, za sebou idúcich, vrstiev ako výškových máp, je možné dosiahnuť dojem vlnenia.

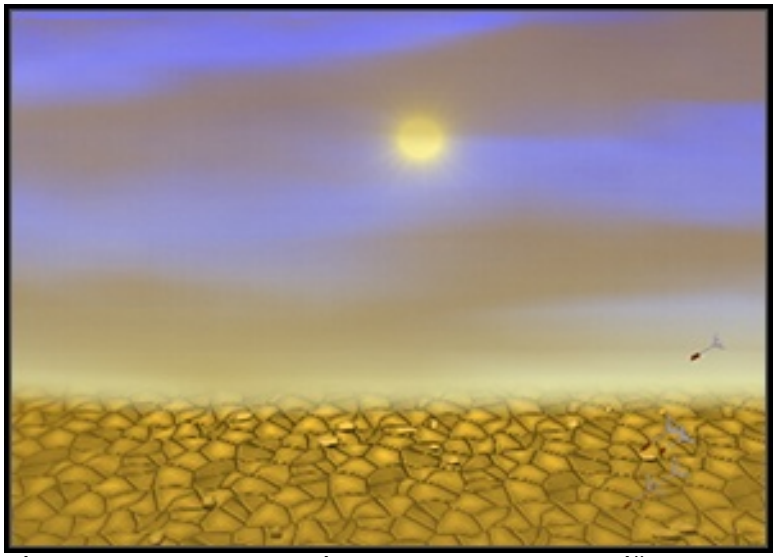
Skutočný vietor sa však „nevlní na mieste“, ale prúdi. Preto sa výšková mapa neprekresľuje na mieste, ale v závislosti od času sa pohybuje v smere zápornej osi `x`.

Vo výslednom intre je vietor vykresľovaný niekoľko sekúnd. Použitie jedného veľkého poľa, ktoré by prechádzalo priestorom, by bolo zbytočné zapĺňanie pamäte, keďže by vždy bola vykreslená len jeho malá časť a na krátky okamih. Preto je výšková mapa rozdelená na dve polovice, ktoré sa medzi sebou prehadzujú podľa aktuálnej pozície (Obrázok 18). V pohľadovom priestore sa tak vždy nachádza minimálne jedna polovica výškovej mapy a mimo vykresľovaného priestoru nedochádza k zbytočným výpočtom. Modrý pás na obrázku 18 znázorňuje miesto, kde sa výškové mapy na seba navzájom napájajú.



Obrázok 18: Ukážka prerozdelenia a prehadzovania výškovej mapy.

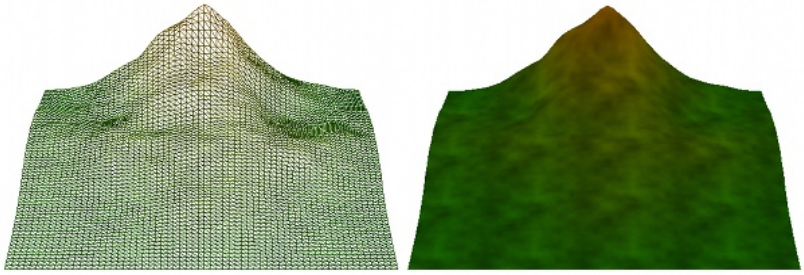
Vyššie uvedeným postupom je vygenerované prúdenie vetra. Druhou dôležitou úlohou je správne ofarbenie výškových máp. Vietor je znázornený prúdením čiastočiek piesku a podľa toho je zafarbený príslušnou farbou. Následným pridaním priehľadnosti sa dosiahne konečný výsledok. Podrobne je postup zafarbenia pomocou shaderu popísaný v kapitole 6.2.5 Vietor.



Obrázok 19: Vietor vytvorený pomocou dynamickej výškovej mapy.

5.7.2 Terén

S cieľom, aby zobrazená zem nevyzerala neprirodzene plocho, je terén zvlnený dvojrozmernou výškovou mapou vytvorenou Perlinovým šumom. Nerovnosti povrchu sú evidentné na konci intra, kde je touto technikou vymodelovaný kopec. V tomto prípade sú vrcholy trojuholníkovej siete ovplyvnené výškovou mapou v závislosti od ich vzdialenosti od kraja. Smerom ku hrane sa terén úplne vyrovná a kopec sa stane dominantným.



Obrázok 20: Použitie Perlinovho šumu pri tvorbe výškovej mapy

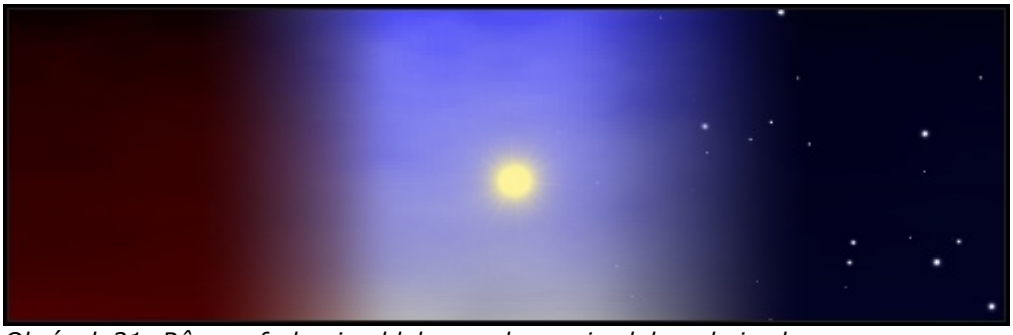
5.8 Obloha

Prevažná časť výsledného dema sa odohráva v exteriéri, kde je potrebné vytvoriť okolitý svet. Jedna z najpoužívanejších techník pre modelovanie otvoreného priestranstva používa tzv. *skybox*, prípadne *skydome*. Ide o metódu, pri ktorej sa okolo kamery vygeneruje kocka (vtedy sa jedná o skybox) alebo guľa (skydome). Na tieto telesá sa následne premietajú vzdialené objekty ako mraky, hory, vzdialené mestá a pod. Často sa pritom negeneruje celá kocka (guľa), ale stačí horná polovica, ktorá zobrazuje oblohu a horizont. Táto technika je rozšírená najmä v počítačových hrách alebo simulátoroch, kde má užívateľ možnosť pohybu. Keďže v deme je pozícia a smer kamery vopred definovaný, teda bez možnosti zásahu, generovanie celého skydomu je zbytočné. Preto je zvolený prístup na vykreslenie oblohy značne odlišný. Využívajú sa pixelové mapy (pixmapy), ktoré sú priamo renderované do framebufferu. Aby bolo možné zobraziť dennú aj nočnú oblohu s plynulým prechodom v priebehu dema, je použitých niekoľko vrstiev s rôznou priehľadnosťou, ktoré sa medzi sebou prekrývajú (Obrázok 21). Jednotlivé použité vrstvy zobrazujú:

- slnko – dvojkanálová pixmapa, kde jeden kanál obsahuje informáciu o priehľadnosti a druhý ukladá farebné hodnoty. Na vytvorenie svetelných lúčov bola použitá exponenciálna funkcia s rôznymi parametrami, čím sa dosiahla rozdielna intenzita lúčov.

- hviezdy – jednokanálová pixmapa definujúca priehľadnosť podľa toho, či sa na určitom mieste nachádza hviezda.
- mraky – jednokanálová pixmapa vytvorená pomocou Perlinovho šumu (kapitola 5.1.1 Perlinov šum)

Pixmapy sú ofarbené pomocou OpenGL funkcií **`glPixelTransfer()`**[22], ktoré definujú spôsob, akým sú jednotlivé hodnoty upravené počas prenosu do bufferu. Pre každú zložku farby je možné nastaviť faktor, ktorým je prichádzajúca hodnota prenásobená a konštanta, ktorá sa má pripočítať. Týmto spôsobom je možné jednokanálové pixmapy ľubovoľne zafarbiť a ušetriť miesto v pamäti.



Obrázok 21: Rôzne ofarbenia oblohy a zobrazenie slnka a hviezd.

Obloha je priamo zapisovaná na výstupný buffer a neprechádza použitými transformáciami projekcie. Preto je nutné presne určiť polohu, kde sa má začať vykresľovať. Aby sa obloha pohybovala v závislosti od polohy a smeru kamery, treba zohľadniť nastavenia projekčnej a modelovo-pohľadovej matice. Pri výpočte sa predpokladá, že zem leží v rovine určenej rovnicou `y=0` a vzdialená orezová rovina je nastavená na vzdialenosť `z_cut`. Úlohou je nájsť priemet bodu `T` – priesečníka osi `z` a zadnej orezovej roviny. Pre získanie výsledných súradníc treba:

- vypočítať inverznú modelovo-pohľadovú maticu $\mathbf{M^{-1}}$ – vykonaním opačných transformácií v obrátenom poradí
- zo vzťahu $\mathbf{(0, y, z\_cut, 1)*M^{-1} = (t_1, 0, t_2, 1)}$ vyjadriť hodnotu **y** a následne bod $\mathbf{T = (t_1, 0, t_2, 1)}$
- pre získanie priemetu bodu **T** vzhľadom na aktuálnu pozíciu kamery je nutné transformovať bod **T** do súradníc kamery - **T'= T*M** a následne do orezových súradníc - **T''= T'*P,** kde **P** je projekčná matica
- finálne súradnice dopočítame z bodu **T''** v závislosti od rozmerov okna a nastavení perspektívy

22 `void` **`glPixelTransfer{fi}`**`(GLenum` *`pname`*`, GLfloat` *`param`*`);`

5.9 Krivky a splajny

Pri modelovaní scény sa často vyskytne potreba vytvoriť hladkú krivku s požadovaným priebehom. Priamo zistiť analytické vyjadrenie takejto krivky je často nemožné, preto sa využívajú interpolačné a aproximačné zadania takejto krivky. Pri interpolačnom zadaní sú definované body, ktorými musí krivka prechádzať. Aproximačné zadanie určuje body, ktoré definujú tvar krivky, nemusia jej však patriť. Podľa vlastností a zvolenej techniky vyčísľovania sa krivky ďalej delia. Spojením viacerých kriviek vzniká splajn. Jednou z najobtiažnejších a najdôležitejších úloh pri vytváraní splajnov je zaistenie hladkého napojenia kriviek.

Jedna z najpoužívanejších aproximačných kriviek – *Bézierova*[23] *krivka* [4]– bola použitá na modelovanie stoniek kvetov. Bézierova krivka určená `(n+1)` bodmi je krivkou n-tého stupňa definovaná vzťahom

$$P(t)=\Sigma_{i=0}^{n} V_i B_{i,n}(t), \quad 0\le t\le 1 \qquad (1)$$

kde V_i sú polohové vektory riadiacich bodov a $B_{i,n}(t)$ sú *Bernsteinove*[24] *polynómy* definované ako

$$B_{i,n}(t)=\binom{n}{i} t^i (1-t)^{n-i}, \quad \binom{0}{0}=1$$

OpenGL ponúka veľmi užitočný nástroj pre vyčísľovanie Bézierových kriviek, tzv. *evaluátory*. Pri používaní evaluátorov je v prvom kroku nevyhnutné popísať generovanú krivku funkciou **`glMap1()`**[25]. Parametre funkcie určujú stupeň krivky, počet riadiacich bodov, hranice krivky a jednotlivé riadiace body. Následne je nutné povoliť jednorozmerný evaluátor pre trojrozmerné vrcholy príkazom **`glEnable`**`(GL_MAP1_VERTEX3)`. Vyčísľovanie bodov krivky pre daný parameter `u`, potom zabezpečuje funkcia **`glEvalCoord1{fd}()`**[26]. Tá je ekvivalentná volaniu funkcie **`glVertex()`**. Evaluátory boli použité pri vyčísľovaní hodnôt kriviek pri počiatočnom raste kvetiny z buniek.

V projekte sa však splajny tvorené Bézierovými krivkami nevyužívajú len na vyčísľovanie polohy vrcholov, ale aj na generovanie parametrov transformácií pri animácii scény. Pre tieto účely bola implementovaná vlastná skupina funkcií pre prácu

23 *Pierre Étienne Bézier (1910 -1999)* - francúzsky inžinier a matematik.

24 *Sergei Natanovich Bernstein (1888 - 1986)* - ruský matematik.

25 `void` **`glMap1{fd}`**`(GLenum` *`target`*`, GLfloat` *`u1`*`, GLfloat` *`u2`*`, GLint` *`stride`*`, GLint` *`order`*`, const GLfloat *`*`points`*`);`

26 `void` **`glEvalCoord1{fd}`**`(GLfloat` *`u`*`);`

so splajnami tvorenými Bézierovými krivkami tretieho stupňa. Z nich je potom zostavená výsledná spojitá krivka. Ku každým dvom za sebou idúcim zadaným bodom sa dopočítavajú dva vnútorné body. Tým sa získajú štyri riadiace vrcholy Bézierovej krivky. Aby bola v hraničných bodoch zabezpečená spojitosť, pri určovaní vnútorných bodov treba zohľadniť predchádzajúci, respektíve nasledujúci bod. V prípade hraničných bodov splajnu dôjde k zdvojeniu. Nech P_1 a P_2 sú krajné body Bézierovej krivky a P_0 (P_3) bod predchádzajúci bodu P_1 (nasledujúci za bodom P_2). Potom vnútorné body R_1 a R_2 je možné získať podľa vzťahov

$$R_1 = P_1 + \frac{(P_2 - P_0)}{4} \qquad R_2 = P_2 + \frac{(P_1 - P_3)}{4}$$

Štyri body P_1, P_2, R_1 a R_2 sú riadiace body Bézierovej krivky tretieho stupňa, ktorej zvyšné body vyčíslime pomocou vzorca (1).

Splajn vytvorený touto metódou je použitý pre určenie polohy vrcholov stoniek a listov kvetov. Tomuto účelu je prispôsobená aj implementovaná funkcia pre výpočet hodnoty splajnu. Pracuje s bodmi, ktorých trojrozmerné súradnice sú uložené v štruktúre `coord`. Tá má tri premenné typu `float` pre uloženie týchto hodnôt. Funkcia vracia pre zadaný parameter *t* príslušnú hodnotu splajnu takisto vo forme štruktúry `coord`. Rovnaká funkcia je použitá pre nastavenie správnych transformácií pri zobrazovaní scény. Bližšie o tom pojednáva kapitola 7.2 Kamera.

6 GLSL a shadre

6.1 OpenGL Shading Language (GLSL)

Do verzie OpenGL 2.0 bolo poradie spracovania vrcholov a fragmentov vo vykresľovacom reťazci presne určené. Programátor nemal možnosť zasiahnuť do poradia, v ktorom sa jednotlivé operácie spracovávania vykonávali. Tento spôsob je známy pod pojmom *fixná pipeline*. OpenGL 2.0 rozširuje pevne daný prístup o programovateľný, ktorý umožňuje riadiť poradie spracovania vrcholov a fragmentov pomocou malých programov – *shaderov.* Pri spracovávaní dát je možné používať vždy len jeden zo spomínaných módov. V prípade použitia shaderov je časť s fixnými funkciami neprístupná.

Pre potreby písania shaderov do aplikácií využívajúcich knižnicu OpenGL bol vyvinutý programovací jazyk nazývaný *OpenGL Shading Language (GLSL)* [3]. GLSL má niekoľko spoločných rysov s programovacím jazykom C++, ako je napríklad konštrukcia cyklov, vytváranie štruktúr a funkcií, či základné operátory. Navyše však obsahuje datové typy a funkcie prispôsobené operáciám s vrcholmi, súradnicami textúr, farbami či maticami. Patria medzi ne napríklad dvoj- až štvor- zložkové vektory, matice, skalárny a vektorový súčin, normalizácia vektoru, interpolácia, dĺžka vektoru, funkcie pre prácu s textúrami atď. Vďaka rôznym modifikátorom, ktoré menia charakteristiky dátových typov, môžu byť hodnoty premenných prístupné iba pre čítanie (modifikátor *const*), zadávané z hlavnej aplikácie (*uniform*), zdieľané medzi jednotlivými shaderami (*varying*) alebo získané z dát uložených pre vrchol v aplikácii OpenGL (*attribute*).

GLSL rozlišuje dva typy shaderov – *vertex* shader pre spracovanie vrcholov a *fragment* shader pre spracovanie fragmentov. Pre každý so spomínaných shaderov sú definované jeho vstupy a výstupy. Ich popis je však mimo tematického rozsahu tejto práce, preto bude zmienený len jediný potrebný výstup. Tým je premenná `gl_Position`, ktorá musí byť zapísaná na výstupe vertex shaderu a obsahuje informácie o homogénnych súradniciach vrcholu po transformácii modelovo-pohľadovou a projektívnou maticou.

Pri vytváraní shaderov sa postupuje nasledovne [1]:

Pre každý objekt shaderu je potrebné:

- vytvoriť objekt shaderu
- skompilovať zdroj shaderu do objektu
- overiť, či bol shader úspešne skompilovaný.

Pre pripojenie viacerých objektov shaderov k programu je potrebné:

- vytvoriť program shaderu

- priradiť objekty shaderu programu
- skompilovať program shaderu
- overiť úspešné prilinkovanie programu
- zapnúť používanie vytvoreného programu.

6.2 Použité shadre

Vo výslednom programe sú použité celkovo štyri shadre. Jeden zabezpečuje vykresľovanie tieňov, skladanie textúr pri bump mapping a hmlu, ďalší je použitý pre vykreslenie blesku a scény osvetlenej bleskom. Shader je použitý aj pri modelovaní vetru a svetelných lúčov. V nasledujúcich kapitolách sú jednotlivé shadre bližšie popísané. Väčšina z nich potrebuje určenie uniformných premenných a parametrov z hlavného programu. Ak sa jedná o dôležité hodnoty a funkcie, sú takisto bližšie popísané. V niektorých prípadoch sa však jedná a menej dôležité konštanty, prípadne zrejmé hodnoty, ktoré nemajú na funkčnosť shaderu veľký vplyv, a preto sú vynechané.

6.2.1 Tiene

Pri vytváraní tieňov s použitím OpenGL a GLSL sa najdôležitejšia časť vykonáva v hlavnom programe. Tento postup je podrobne rozpísaný v kapitole 5.3 Tiene. V nasledujúcej časti je popísaná len časť odohrávajúca sa vo vertex a fragment shaderi. Predpokladá sa teda, že je vytvorená hĺbkovú mapa, ktorú využíva vstavaná funkcia fragment shaderu `vec4` **`shadow2DProj`**`(sampler2DShadow` *`sampler`*`,` `vec4` *`coord`* `[,` `float` *`bias`*`])`. Tá porovnáva hodnotu danú súradnicami *`coord`* s hodnotami v hĺbkovej mape určenej parametrom *`sampler`*. Samotné porovnanie je medzi treťou zložkou *`coord.p`* a hodnotou uloženou v hĺbkovej mape na súradniciach určených prvými dvoma zložkami premennej *`coord`*. Keďže hĺbková mapa má často jemnejšie vzorkovanie ako buffer, do ktorého sa zapisuje, pri určovaní textúrovacích súradníc treba dávať pozor na správne indexovanie. Návratová hodnota funkcie **`shadow2DProj`** je 1.0 ak je fragment „osvietený“ a 0.0 ak sa nachádza v tieni. Toto striktné rozhodnutie vedie k vytvorenie veľmi ostrých tieňov. Antialiasing hrán je však v tomto prípade veľmi jednoduchý – výsledná hodnota pre každý fragment sa získa ako priemerná hodnota z okolia. Čím väčšie okolie, tým sa získa hladší prechod, ale súčasne sa zvýši aj výpočtová náročnosť algoritmu. Ďalším nežiaducim artefaktom, spôsobeným porovnávaním hodnôt, je tzv. samozatienenie. Pomocou orezávania privrátených strán pri vytváraní hĺbkovej mapy sa síce tento jav na privrátených stranách objektu odstránil, na odvrátených stranách ho však stále možno

pozorovať. Preto je pri porovnávaní hodnôt pripočítaná malá hodnota, ktorá zaručí minimálny rozdiel pri porovnaní.

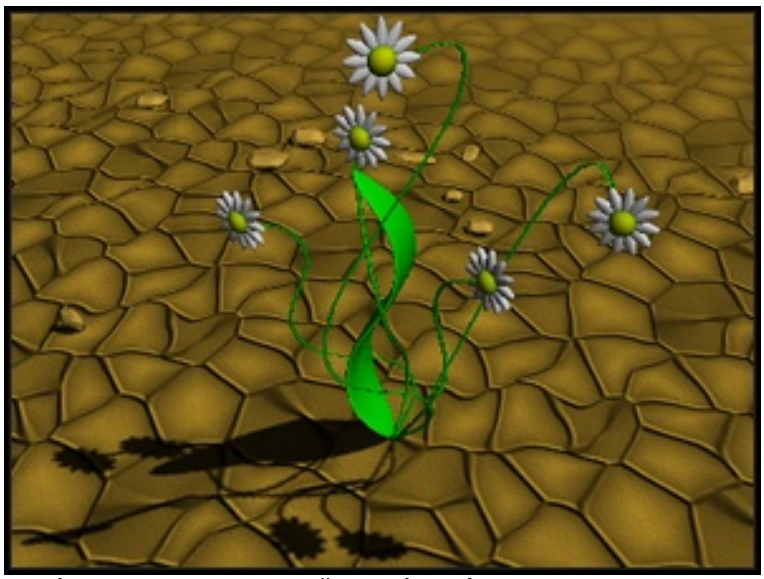
Vo fragment shaderi nakoniec bude funkcia pre porovnávanie s hĺbkovou mapou, ktorá zohľadňuje *offset* a posun v hĺbkovej mape:

```
float lookup(vec2 offSet) {
        return shadow2DProj(ShadowMap, ShadowCoord + vec4(offSet.x * xPixelOffset *
        ShadowCoord.w, offSet.y * yPixelOffset * ShadowCoord.w, 0.05, 0.0) ).w;"
}
```

Spomínaný antialiasing na hranách tieňa zabezpečený spriemerovaním hodnôt v okolí vykresľovaného fragmentu je implementovaný nasledovne:

```
for (float y = -1.5 ; y <=1.5 ; y+=1.0)
        for (float x = -1.5 ; x <=1.5 ; x+=1.0)
                shadow += lookup(vec2(x,y));
shadow /= 16.0;
```

Hodnota `shadow` sa neskôr využije pri určení finálnej hodnoty výstupnej premennej fragment shaderu `gl_FragColor`.



Obrázok 22: Kvetina a tieň, ktorý vrhá.

6.2.2 Bump mapping

Teória emboss bump mapping je priblížená v kapitole 5.1.3 Bump mapping. Hlavný program v tomto prípade zabezpečuje pripojenie všetkých troch textúr - bump mapy, inverznej bump mapy a hlavnej textúry, ktorá nesie farebnú informáciu. Výpočet posunu inverznej bump mapy v závislosti od pozície svetla a kamery a predanie textúrovacích súradníc všetkých troch textúr pre jednotlivé vrcholy takisto zabezpečuje OpenGL aplikácia. Vertex shader v tomto prípade prepojí súradnice textúr s prislúchajúcimi textúrovacími jednotkami:

```
gl_TexCoord[0] = gl_MultiTexCoord0;
gl_TexCoord[1] = gl_MultiTexCoord1;
gl_TexCoord[2] = gl_MultiTexCoord2;
```

Vo fragment shaderi sú potom hodnoty textúr zmiešané tak, aby bol dosiahnutý požadovaný výsledok. Hodnoty získané z bump máp sa sčítajú, čím sa vytvorí dojem reliéfu, a následne sa fragment ofarbí vynásobením hodnotou z farbenej textúry:

```
vec3 color1 = texture2D(texBump, gl_TexCoord[0].xy);
vec3 color2 = texture2D(texInvBump, gl_TexCoord[1].xy);
vec3 color3 = texture2D(texColor, gl_TexCoord[2].xy);
vec3 color  = (color1+color2)*color3;
```

Hodnota `color` sa ďalej využije pri záverečnom výpočte výstupnej premennej fragment shadru `gl_FragColor`.

6.2.3 Hmla

Efekt hmly je použitý s cieľom, aby medzi zemou na horizonte a oblohou nevznikal ostrý prechod. Vďaka nej sa zem v diaľke stráca a plynule prechádza do oblohy. Keďže povrch zeme je otextúrovaný pomocou shadrov, hmlu nie je možné použiť jednoduchým volaním z hlavného programu. Fixné funkcie sú totiž pri použití shaderov blokované a hmla by tak nebola pri výpočte zohľadnená. Namiesto toho je použitý shader, ktorý vo vertex shaderi predá súradnice hmly priradením:

```
gl_FogFragCoord = gl_FogCoord;
```

Vo fragment shaderi je potom pomocou vstavaných premenných `gl_Fog.scale`, `gl_Fog.end` a `gl_Fog.color` vypočítaná hodnota hmly s lineárnym postupom vzhľadom na pozíciu voči kamere. Výsledná hodnota je orezaná a následne použitá pri miešaní farby

fragmentu s farbou hmly [16]. Keďže obloha na pozadí nemá všade rovnakú hodnotu, narozdiel od farby hmly, ktorá je konštantná, je pridaná priehľadnosť určená hodnotou intenzity hmly. Spojenie všetkých spomínaných funkcií potom vyzerá nasledovne:

```
float fog = (gl_Fog.end - gl_FragCoord.z / gl_FragCoord.w) * gl_Fog.scale;
fog = clamp(fog, 0.0, 1.0);
gl_FragColor = vec4(mix(vec3(gl_Fog.color), color, fog), fog);
```

Parametre hmly sa nastavia v hlavnom programe volaním funkcií **glFog()**[27]. Farba hmly je dynamicky menená vzhľadom na farbu oblohy. Tá sa získa načítaním spodného bodu pixmapy použitej ako obloha pomocou funkcie **glReadPixels()**[28].

6.2.4 Šedotónové zobrazenie a blesk

V prvej polovici intra je zobrazená búrka. Pre umocnenie dojmu sú k búrke pridané blesky. Blesk je tvorený jednokanálovým rastrom, ktorý určuje hodnotu pre priehľadnosť a farbu, ktorou má byť daný bod rastru ofarbený. Táto hodnota klesá smerom od stredu rastru k okrajom a od horného kraja smerom nadol. Následne je blesk upravený pomocou Perlinovho šumu (Obrázok 4). Počas vykreslenia blesku je celá scéna zobrazená v odtieňoch šedej, akoby bola bleskom osvietená. Keďže sa v scéne vyskytujú otextúrované aj neotextúrované objekty, shader pracuje v niekoľkých módoch, pričom farebná hodnota je získaná vždy rôznym spôsobom. Pre jednotlivé prípady sa farebná zložka získa nasledovne:

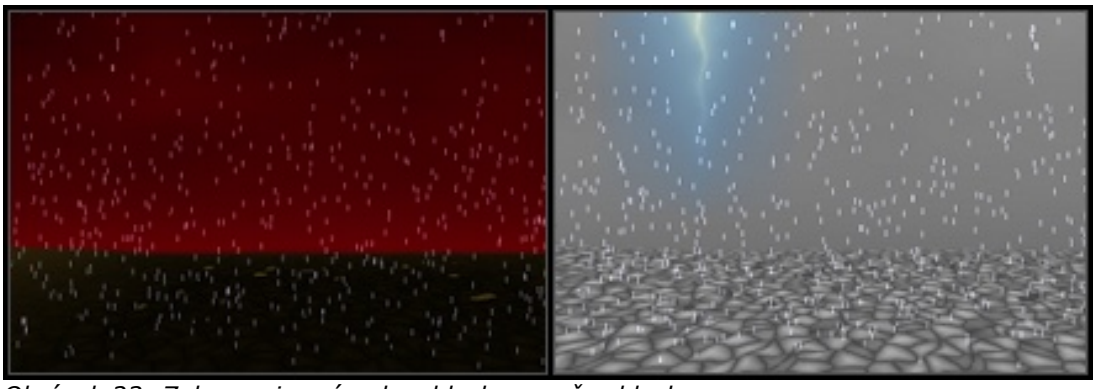
- otextúrované objekty (farebná hodnota je prevzatá z textúry) :
  ```
  color = texture2D(TexSampler, gl_TexCoord[0].xy);
  ```
- neotextúrované objekty (farba prichádzajúceho fragmentu, červená zložka je prevrátená, keďže týmto spôsobom je ofarbená obloha, kde sa nachádzajú tmavé mraky na červenom pozadí a vo výsledku je vhodné mať osvietené oblaky):
  ```
  color = vec4(1.0 - gl_Color.r, gl_Color.gba);
  ```
- blesk (blesk má farebný prechod od svetlo modrej v strede po svetlo žltú na krajoch, hodnotu pre prechod sa získa z alfa zložky fragmentu):
  ```
  color = mix(vec3(0.0,0.5,1.0), vec3(1.0,1.0,0.8), gl_Color.a);
  ```

27 `void glFog{if}(GLenum pname, TYPE param); void glFog{if}v(GLenum pname, TYPE *params);`

28 `void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);`

Výsledná farebná hodnota pre objekty je nakoniec prevedená do šedotónového zobrazenia podľa vzťahu `R*0.299 + G*0.587 + B*0.114`, kde `R`, `G`, `B` určujú podiel červenej, modrej a zelenej farebnej zložky. Využíva sa pri tom vstavaná funkcia shaderu pre výpočet skalárneho súčinu:

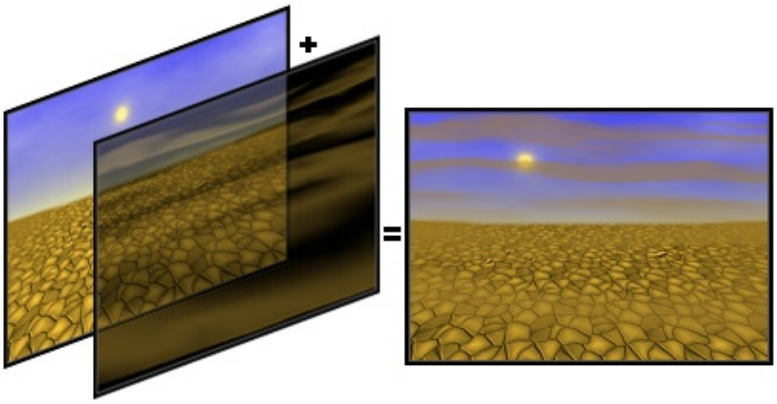
```
color = dot(color, vec3(0.299, 0.587, 0.114));
gl_FragColor = vec4(color, gl_Color.a);
```



Obrázok 23: Zobrazenie scény bez blesku a počas blesku.

6.2.5 Vietor

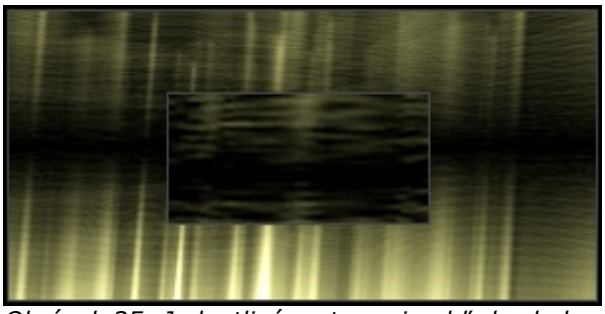
Vietor je vymodelovaný pomocou výškovej mapy a trojrozmerného Perlinovho šumu. Tieto techniky sú bližšie vysvetleného v kapitolách 5.1.1 Perlinov šum a 5.7 Výšková mapa. Shader v tomto prípade zabezpečuje natočenie výškovej mapy smerom ku kamere (vertex shader) a ofarbenie (fragment shader). RGB zložky farby sú konštantne nastavené pre celý objekt, mení sa len alfa zložka, teda priehľadnosť. Tá je vypočítaná v závislosti od pôvodnej výšky fragmentu pred natočením výškovej mapy a jednotlivé prúdy vetra sú dosiahnuté pomocou funkcie kosínus.



Obrázok 24: Ofarbenie výškovej mapy vo fragment shadri a výsledná scéna s vetrom.

6.2.6 Svetelné lúče

V úvode intra je použitý efekt presvitajúcich svetelných lúčov. Ide o niekoľko šedotónových textúr preložených cez seba, pričom s klesajúcou výškou sa zväčšuje oblasť, na ktorú je textúra namapovaná. Tým sa vytvorí dojem, akoby sa lúče rozptyľovali. Kvôli zoradeniu jednotlivých textúr možno pri pohľade zboku vidieť jednotlivé vrstvy (Obrázok 25). Vhodným nastavením kamery sa však dá tomuto nežiadúcemu artefaktu vyhnúť. Použitá je textúra, ktorá je aplikovaná na vytvorenie popraskanej zeme.

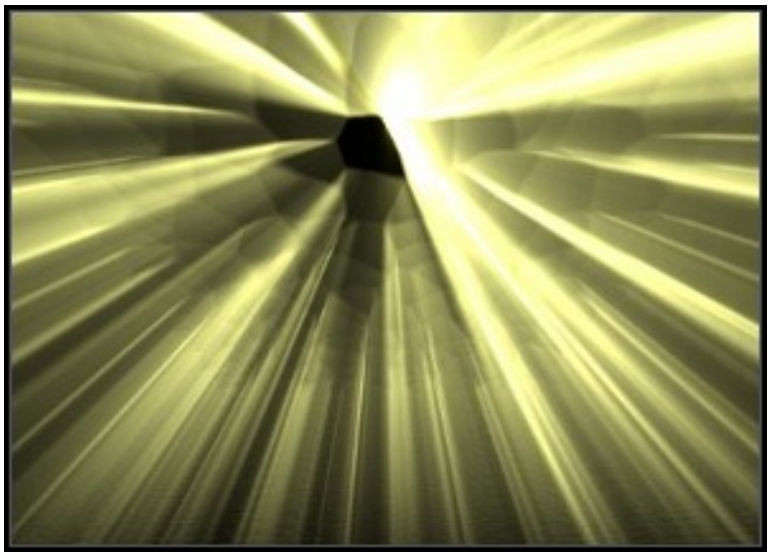


Obrázok 25: Jednotlivé vrstvy pri pohľade zboku.

Vertex shader sa v tomto prípade stará len o priradenie textúrovacích súradníc. Vo fragment shaderi je textúra ofarbená a je jej nastavená príslušná priehľadnosť:

```
vec3 color = texture2D(TexSampler, gl_TexCoord[0].xy);
gl_FragColor = vec4(color.r, color.g, color.b*0.6, 0.02);
```

Pôvodne boli svetelné lúče vytvorené pomocou akumulačného bufferu. Tento prístup však dosahoval veľmi nízky framerate. S využitím shaderu sa framerate podarilo niekoľkonásobne zvýšiť.



Obrázok 26: Svetelné lúče vytvorené vrstvami ofarbenými vo fragment shadri.

7 Hudobný podklad a kamera

7.1 Hudobný podklad

Neoddeliteľnou súčasťou intra je hudobný podklad. Pomáha dotvárať výsledný dojem a pri správnej synchronizácii s kamerou a jednotlivými scénami môže niekoľkonásobne zvýšiť hodnotu výsledného diela. Samozrejme to platí aj naopak a v prípade nevhodného výberu skladby sa graficky veľmi dobre zvládnuté dielo môže znehodnotiť.

Bežne používané audio formáty, ako napríklad *mp3*[29] alebo *wav*[30], dosahujú pri priemerne dlhých skladbách veľkosť niekoľko megabytov. Je samozrejmé, že kvôli veľkosti nemôžu byť takéto statické dáta pridané k výslednému programu. Ďalšou možnosť je využitie formátu *MIDI*[31], pri ktorom sa ukladajú len jednotlivé noty. Takto vytvorená hudba je však čisto závislá na hardwari a preto programátor nemá plnú kontrolu nad výsledným zvukom. To je dôvod, prečo už dnes nie je tento spôsob príliš rozšírený a používa sa len ojedinele, a to pri intrách s veľkosť do 4kB. V dnešnej dobe sú najpoužívanejšie tzv. *syntetizéry*, ktoré pomocou matematických funkcií dokážu vygenerovať jednotlivé hudobné efekty a zostaviť z nich skladbu, ktorá sa svojou kvalitou vyrovná najpoužívanejším formátom, pričom si zachová minimálnu veľkosť. Existuje viacero kvalitných syntetizérov, no pravdepodobne najpoužívanejším je *V2 Synthesizer System* [21]. Jeho autorom je *Tammo Hinrichs*, člen nemeckej demoskupiny *Farbraush* [18]. Syntetizér pracuje s vlastným formátom *v2m.* Ten je navrhnutý tak, aby bol čo najlepšie skomprimovateľný a vo výslednom súbore zaberal približne 5 až 10 kB. Syntetizér využíva súčasť knižnice *DirectX* pre prácu so zvukom – *DirectSound*. Súčasťou syntetizéru sú aj pluginy do programov *VST Instrument* a *Buzz Machine* pre komponovanie vlastných skladieb a plugin do programu *Winamp* pre prehrávanie formátu *v2m.*

Vzhľadom na náročnosť skomponovania vlastného hudobného doprovodu, ktoré vyžaduje vyšší stupeň hudobného nadania, bola v práci využitá voľne dostupná skladba od autora vystupujúceho pod prezývkou *Soft Maniac* s názvom *Follow The Light II*.

Pre import, prehrávanie a ukončenie slúži štruktúra `V2MPlayer` a jej metódy **`Init()`**, **`Open()`**, **`Play()`** a **`Close()`**.

29 mp3 - MPEG-1 Audio Layer 3.

30 wav - Waveform Audio File Format.

31 MIDI - Musical Instrument Digital Interface.

7.2 Kamera

Pre grafické intro je dôležité, aby scéna nepôsobila príliš staticky. Preto je nutné okrem animácie samotných objektov zabezpečiť aj pohyb kamery okolo scény. Je veľmi výhodné, ak je pohyb kamery koordinovaný s hudobným doprovodom. Keďže však bol výber možných skladieb obmedzený, nie vždy sa podarilo dosiahnuť požadovaný výsledok. Pri riešení problematiky snímania scény je nutné vyriešiť dva zásadné problémy – nastavenie pohľadovej matice určujúcej polohu kamery a smer pohľadu a generovanie krivky, po ktorej sa bude kamera plynule pohybovať.

Pri nastavení pozície kamery a scény bolo možné využiť dve možnosti. Prvou je nastavenie pomocou funkcie `void` **`gluLookAt`**`(GLdouble` *`eyeX`*`, GLdouble` *`eyeY`*`, GLdouble` *`eyeZ`*`, GLdouble` *`centerX`*`, GLdouble` *`centerY`*`, GLdouble` *`centerZ`*`, GLdouble` *`upX`*`, GLdouble` *`upY`*`, GLdouble` *`upZ`*`)`, kde prvé tri parametre určujú pozíciu kamery, nasledujúce tri súradnice bodu, na ktorý je kamera zameraná a posledné tri určujú vektor smerujúci nahor od kamery. Druhou možnosťou je skladanie translačných a rotačných transformácií volaním funkcií `void` **`glTranslatef`**`(GLfloat` *`x`*`, GLfloat` *`y`*`, GLfloat` *`z`*`)` a `void` **`glRotatef`**`(GLfloat` *`angle`*`, GLfloat` *`x`*`, GLfloat` *`y`*`, GLfloat` *`z`*`)`. Výhodou prvej možnosti je najmä priama kontrola pozície kamery a smeru pohľadu zadaním presných priestorových súradníc. Vo výpočtoch vykonaných behom programu však využívam inverzné zobrazenie. Výpočet inverznej modelovo-pohľadovej matice určenej funkciou **`gluLookAt()`** je však pomerne náročný a zdĺhavý. Inverznú maticu je pritom potrebné počítať pri každom prekreslení scény, pre správne nastavenie pozície oblohy. Tu sa ukáže najväčšia výhoda prístupu so skladaním transformácií. Stačí resetovať modelovo-pohľadovú maticu (volaním funkcie **`glLoadIdentity()`**), vykonať inverzné transformácie v opačnom poradí ako pri zobrazovaní a načítať modelovo-pohľadovú maticu do zvolenej premennej `matrix` pomocou funkcie **`glGetFloatv`**`(GL_MODELVIEW_MATRIX, matrix)`. Preto bola v práci využitá druhá možnosť. Transformácie, ktoré určujú polohu, sú (v poradí):

- posun v smere osi `z`
- rotácia okolo osi `x`
- rotácia okolo osi `y`
- prídavné posuny v smere osí `x` a `y`.

Druhou spomínanou úlohou pri riešení pohybu je určenie polohy kamery alebo v tomto prípade parametrov transformácií v danom čase. Je samozrejmé, že tieto hodnoty nemožno priamo definovať pre každý okamih. Preto sa využíva interpolácia medzi

niekoľkými, presne určenými riadiacimi bodmi. Pri použití jednoduchej lineárnej interpolácie je však pohyb veľmi trhaný. Pre dosiahnutie plynulého pohybu sa preloží bodmi spojitá krivka. Problematika interpolačných a aproximačných kriviek je bližšie diskutovaná v kapitole 5.9 Krivky a splajny. V tomto prípade je takisto použitý splajn tvorený Bézierovými krivkami tretieho stupňa, ktorý je popísaný v spomínanej kapitole. Pretože bol použitý algoritmus pôvodne vytvorený na posun vrcholov objektov, návratovou hodnotou funkcie pre určenie bodu krivky je štruktúra obsahujúca tri hodnoty (súradnice v priestore). Pri pohybe kamery sa však nepracuje so súradnicami bodov, ale s hodnotami určujúcimi jednotlivé transformácie. Presne povedané s posunom v smere osi z a rotáciami okolo osí x a y. Tieto tri hodnoty sú presne určené v uzlových bodoch, medzi ktorými sa vykoná spomínaná interpolácia. Riadiaci bod je definovaný pre každé štyri sekundy. Poloha na krivke sa potom vypočíta pre parameter získaný na základe aktuálneho času. Tým je zaručené, že pohyb kamery je čisto závislý na čase a v prípade nižšieho frameratu sa kamera pohybuje po väčších úsekoch. Zvyšné dva posuny v smere osí x a y sa vyčísľujú individuálne v závislosti od práve vykresľovanej scény. Využíva sa pri tom lineárna i kosínusová interpolácia.

8 Kompresia a výsledný program

8.1 Štruktúra programu

8.1.1 Organizácia súborov

Súbory výsledného programu možno podľa funkcií, ktoré obsahujú, rozdeliť do nasledujúcich skupín:

- modelovacie – obsahujú algoritmy popísané v kapitole 5 Použité techniky. Pre lepšiu prehľadnosť sú pomenované *alg_popis.cpp*. Kde *popis* charakterizuje obsah súboru. Ako príklad možno uviesť súbor, v ktorom sú implementované funkcie pre modelovanie kvetiny s názvom *alg_flower.cpp*.
- pomocné – súbory s funkciami, ktoré priamo nevykresľujú objekty ale sú pre prácu nevyhnutné. Medzi ne možno zaradiť matematické a alokačné funkcie, funkcie pre pohyb kamery, riadenie vykresľovania scény, ovládanie časovača či funkcie slúžiace na pripojenie potrebných rozšírení knižnice OpenGL. Tieto súbory začínajú prefixom *sys_*.
- dáta – jediný súbor obsahujúci vstupné dáta ako súradnice riadiacich vrcholov objektov, uzlové body pre trajektóriu pohybu kamery, hodnoty vykresľovaných bitmáp a pod.
- shadre – obsahom sú zdrojové kódy shaderov a funkcie pre ich kompiláciu.
- hudba – súbory syntetizéru *v2m*.
- systémové – obsahujú vstupnú funkciu programu, vytvorenie okna, zaregistrovanie hlavných tried a pod.

8.1.2 Riadenie behu programu

Aby sa často využívané hodnoty, ako napríklad aktuálny čas, nemuseli predávať každej funkcii vo forme vstupných parametrov, bola vytvorená špeciálna štruktúra `SceneSettings` s nasledujúcimi premennými:

```
typedef struct {
	GLfloat fogColor[4];		//farba hmly
	GLfloat mainLight[3];		//pozícia svetla
```

```
        GLfloat lightIntesity;      //intenzita svetla
        GLfloat zCut;               //vzdialenosť zadnej orezovej roviny
        float  time;                //uplynutý čas v sekundách
        long startTime;             //počiatočný čas
        float day_long;             //dĺžka dňa (interval striedania deň/noc)
        GLubyte scene;              //ID práve vykresľovanej scény
        GLfloat xrot, yrot;         //natočenie scény
        GLfloat xmove, ymove, zmove; //posun scény
        float sky_base[3];          //priemet bodu na horizonte
}SceneSettings;
```

Pomocou globálnej premennej `SceneSettings` *`sceneSet`* sú potom uvedené hodnoty prístupné v každej časti programu.

Väčšina použitých techník vyžaduje prvotnú inicializáciu. Je potrebné vygenerovať textúry pomocou Perlinovho šumu, objekty či pixmapy použité na vykresľovanie oblohy. Keďže sú tieto úlohy časovo náročnejšie, inicializácia prebieha na začiatku programu. Počas inicializácie sa vykresľuje tzv. *progress bar*, ktorý informuje o stave inicializácie. Samotný priebeh programu potom nie je zdržovaný dodatočným generovaním potrebných dát. Pri ukončení programu dôjde k uvoľneniu všetkej použitej pamäte.

V priebehu intra sa vystrieda desať hlavných scén. Pre každú scénu je vytvorená funkcia, ktorá na vykreslenie príslušných objektov volá odpovedajúce funkcie. Pri zmene scény je často potrebné zmeniť aktuálne nastavenie stavového stroja. Takouto zmenou je napríklad posun zadnej orezovej roviny alebo povolenie a zakázanie hmly. Pre riadenie práve používanej funkcie a zmenu nastavení slúžia dve polia ukazovateľov na funkcie:

```
typedef void (*ptrFuncArray)();     //definovaný typ ukazovateľa na funkciu
ptrFuncArray mainRenderFunc[10];    //ukazovatele na funkcie vykresľovania
ptrFuncArray mainSwitchFunc[10];    //ukazovatele na funkcie pre zmenu nastavení
                                    //pri zmene scény
```

Rozhodnutie, ktorá funkcia sa má použiť, záleží na aktuálnom čase behu intra. Trvanie jednotlivých scén je definované pomocou poľa `float` *`main_timePoints[]`*, v ktorom sú uložené časové hodnoty, pri ktorých dôjde k zmene scény. Pri každom prekreslení scény sa kontroluje, pre ktorú najvyššiu hodnotu `i` ešte platí

```
sceneSet.time > main_timePoints[i]
```

Ak je `i` rôzne, ako hodnota uložená v premennej `sceneSet.scene`, teda došlo k zmene vykresľovanej funkcie, zavolá sa `mainSwitchFunc[i]` pre zmenu nastavení a hodnota `sceneSet.scene` sa nastaví na `i`. Keďže pohyb kamery je definovaný zvlášť pre každú scénu, je takisto potrebné načítať nové riadiace body určujúce trajektóriu kamery. Nakoniec sa zavolá príslušná vykresľovacia funkcia `mainRenderFunc[sceneSet.scene]`.

8.2 Kompresia

Aj napriek použitiu mnohých minimalistických techník a optimálnemu nastaveniu kompilátora, veľkosť výsledného programu presahuje maximálnych povolených 65 536B. Dosiahnutú hodnotu je však možné ďalej znížiť prostredníctvom komprimačných nástrojov pre *exe*[32] súbory, ktoré z finálneho programu odstránia redundantné dáta, skomprimujú kód a pridajú dekomprimačný nástroj. Po spustení dôjde najprv k rozbaleniu programu do pamäte, a následnému spusteniu samotného programu. Časové oneskorenie spôsobené úvodnou dekompresiu je, pri takom malom programe ako je grafické intro, nepostrehnuteľné. Nevýhodou týchto komprimačných nástrojov je náchylnosť antivírových programov hlásiť nebezpečný obsah a blokovať takto zabalené programy pri pokuse o spustenie. Je to spôsobené tým, že veľké množstvo škodlivého softwaru sa maskuje práve týmto spôsobom. Kompresia spustiteľných programov sa používala v minulosti, v dôsledku obmedzenej veľkosti miesta na disku a médiách. Dnes sa používajú len v špeciálnych programoch, ako sú práve intrá a bohužiaľ aj vírusy.

Rozličné nástroje sa medzi sebou líšia použitými algoritmami kompresie a špecializáciou pre komprimované programy. Testované kompresné nástroje boli:

- *UPX* (*Ultimate Packer for eXecutable*) – jeden z najznámejších exe-packerov, rozšírený najmä vďaka jeho voľnej distribúcii, podpore mnohých formátov a operačných systémov [23].
- *WinUpack* – voľne šíriteľný komprimačný nástroj [24].
- *MEW* – komprimačný nástroj určený práve pre programy ako sú grafické intrá a demá [25].
- *FSG* (*Fast Small Good*) – takisto komprimačný nástroj vhodný pre menšie programy [26].
- *kkrunchy* – kompresný nástroj od demoskupiny Farbrausch, ktorý je prispôsobený charakteristikám 64k intra [22].

32 exe (*executable file*) - prípona určená pre spustiteľný súbor v prostredí operačných systémov ako *DOS* alebo *MS Windows*.

Program mal pred komprimáciou veľkosť 97 792 B. Dosiahnuté komprimačné výsledky zobrazuje Tabuľka 1.

nástroj	verzia	konečná veľkosť
kkrunchy	0.23 alpha 2	32 256 bytov
WinUpack	0.39 final	37 032 bytov
MEW	SE v1.2	38 814 bytov
FSG	v2.0	41 085 bytov
UPX	3.05w	41 984 bytov

Tabuľka 1: Dosiahnuté výsledky po použití komprimácie.

Konečné veľkosti dosahujú veľmi porovnateľné hodnoty. Pod hraničných 64kB dokázali program skomprimovať všetky packery. Z dosiahnutých výsledkov je však zrejmé, že najlepšiu kompresiu dosiahol program *kkrunchy*, špecializovaný práve na intrá do 64kB, ktorý je v tejto oblasti považovaný za jeden z najlepších programov. Najhorší kompresný pomer dosiahol nástroj UPX, pravdepodobne kvôli jeho univerzálnosti.

8.3 Porovnania

Ako už bolo v úvode spomenuté, je pomerne ťažké zaistiť prenositeľnosť intra. Hoci je program navrhnutý pre maximálnu kompatibilitu, nie všetky grafické karty sú schopné dosahovať rovnaké výsledky. Pri vývoji bola použitá grafická karta *GeForce GT 130M* od spoločnosti *nVidia*.

Najväčším obmedzením sa javí potreba OpenGL verzie 2.0 a vyššej. Hoci aktuálnou je verzia 4.0, mnohé z testovaných kariet toto kritérium nespĺňali. Patrili medi ne najmä integrované grafické karty ako *Intel 945GM*, *Intel X3100* či *Intel 82945G*. Ďalším problémom, ktorý sa vyskytol pri testovaní bol menší počet kariet od spoločnosti *ATI*. Dve z testovaných kariet nepodporovali požadovanú verziu OpenGL. Jediná karta s dostačujúcou verziou však nedokázala vykresliť scény s použitými shadrami. Najlepšie výsledky dosahovali grafické karty od *nVidie*. Konečné výsledky a priemerný framerate vyjadrený v počte snímkov za sekundu (*frames per second FPS*) zobrazuje Tabuľka 2. Porovnávané boli všetky tri možnosti nastavenia rozlíšenia pri zapnutom a vypnutom zvuku. V Tabuľke 3 sú hodnoty maximálneho a minimálneho frameratu nameraného pri nastavení rozlíšenia na 800x600 pixelov a pri zapnutom hudobnom doprovode. Namerané

hodnoty môžu poskytnúť náhľad na potreby a náročnosť intra, nemožno ich však brať ako záväzné, najmä kvôli množstve faktorov, ktoré ovplyvňujú vykonávanie programu.

	640x480		800x600		1024x768	
grafická karta	**ON**	**OFF**	**ON**	**OFF**	**ON**	**OFF**
nVidia GeForce GT 130M	51.0	51.4	47.6	48.4	43.4	44.2
nVidia GeForce 9650M GT	50.1	50.7	48.5	48.9	42.9	43.5
nVidia GeForce 8400M GS	43.6	43.3	40.8	41.5	37.5	38.1
nVidia GeForce 8400M G	31.5	33.9	31.8	31.1	29.8	30.2

Tabuľka 2: Dosiahnuté výsledky pri rôznych grafických kartách.

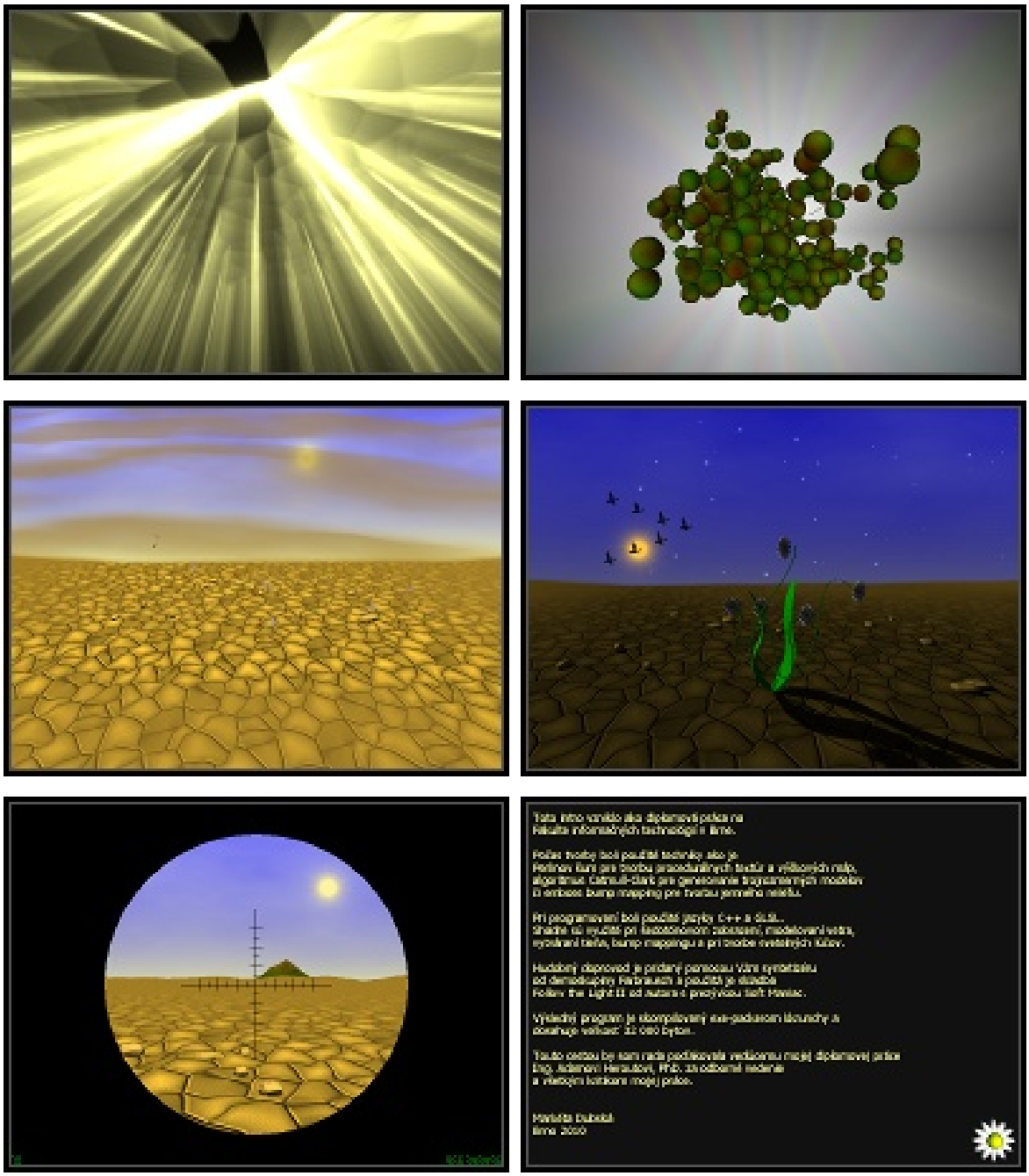
	min	**max**
nVidia GeForce GT 130M	16	61
nVidia GeForce 9650M GT	16	61
nVidia GeForce 8400M GS	14	61
nVidia GeForce 8400M G	7	60

Tabuľka 3: Maximálny a minimálny počet FPS pri zapnutom zvuku a rozlíšení 800x600.

Z dosiahnutých výsledkov vyplýva, že priemerný framerate dosahuje celkom vysoké hodnoty. Nevyhovujúcou sa javí minimálna nameraná hodnota FPS na niektorých grafických kartách. Hodnoty pod 15 FPS sú už veľmi nízke a spôsobujú trhaný pohyb. Náročnou scénou je najmä rast kvetiny, kde sú vrcholy objektov posúvané podľa Bézierovych kriviek. Pri niektorým grafických kartách bol nízky framerate nameraný pri scéne s dažďom, kde sú vykresľované dažďové kvapky pomocou point spritov.

8.4 Screenshoty

Nasledujúce obrázky zobrazujú kľúčové scény intra.



Obrázok 27: Snímky z výsledného dema.

9 Záver

Cieľom diplomovej práce bolo naštudovanie problematiky tvorby grafického dema a techník používaných v tejto oblasti a vytvorenie vlastného diela. V úvode bolo nevyhnutné zoznámiť sa pojmami ako grafické intro a demo. Napriek faktu, že odbornej literatúry venujúcej sa tejto problematike je nedostatok, na internete sa dá nájsť množstvo kvalitných a hodnotných zdrojov. Chýbajúce publikácie možno vysvetliť nelegálnym počiatkom demoscény, ale aj jej súčasnému rýchlemu rozvoju.

V práci sú priblížené minimalistické techniky pre tvorbu objektov a textúr, ako aj použitie jazyka GLSL. Súčasným trendom vývoja grafických knižníc, medzi ktoré patrí aj OpenGL, je úplne odstránenie fixnej pipeline pri vykresľovaní a jej plné nahradenie shaderami. Použitie GLSL síce spôsobilo obmedzenie výsledného intra len na stroje s minimálnou verziou OpenGL 2.0, keďže sa však použitie shaderov stáva štandardom, toto obmedzenie sa týka len starších grafických kariet.

Požadovaný limit 65 536 bytov sa podarilo splniť s rezervou dostačujúcou pre pokračovanie projektu. Zvyšných takmer 32kB voľného miesta by sa dalo využiť pre pridanie ďalších efektov a scén na zvýšenie umeleckej hodnoty. Hlavným problémom teda nebola minimalizácia algoritmov, ale požiadavok, aby sa všetky udalosti odohrávali v reálnom čase a animácia bola plynulá.

Jedným z možných cieľov pokračovania tejto práce je zloženie vlastného hudobného podkladu. Použitím už vytvorenej skladby sa nikdy nedá dosiahnuť požadovaná synchronizácia a výsledný dojem.

V závere možno konštatovať, že práca na projekte bola celkovo veľmi zaujímavá a zahŕňala okrem naštudovania problematiky z oblasti grafiky a OpenGL, aj nové poznatky z matematiky, programovacieho jazyka C++ či nastavenia kompilátoru.

Literatúra

[1] Shreiner, D., Woo, M., Neider, J., Davis, T.: *OpenGL Průvodce programátora.* Brno, Computer Press, 2006, ISBN 80-251-1275-6

[2] Ebert, D. S., Musgrave, F.K., Peachey, D., Perlin K., Worley, S.: *Texturing and Modeling: A Procedural Approach, Third Edition.* USA, Morgan Kaufmann Publishers, 2003, ISBN 1-55860-848-6

[3] Rost, J. Randi: *OpenGL Shading Language, Second Edition.* Addison Wesley Professional, 2006, ISBN 0-321-33489-2

[4] Ferko, A., Ružický, E.: *Počítačová grafika a spracovanie obrazu.* Bratislava, SAPIENTA, 1995, ISBN 80-967180-2-9

[5] OpenGL – The Industry Standard for High Performance Graphics [online]. Dostupné z WWW: <http://www.opengl.org/>

[6] Microsoft Corporation: Compiler Options (C++) [online]. 2010 [cit. 1.5.2010]: Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/9s7c9wdw%28v=VS.80%29.aspx>

[7] The Yasm Modular Assembler Project [online]. [cit. 1.5.2010]: Dostupné z WWW: <http://www.tortall.net/projects/yasm/>

[8] Filiatreault, R.: *Simply FPU* [online]. 2.1.2005 [cit 19.5.2010]: Dostupné z WWW: <http://www.website.masmforum.com/tutorials/fptute/index.html>

[9] Catmull, E., Clark, J.: Recursively generated B-spline surfaces on arbitrary topological meshes, *Computer aided design,* 1978, vol. 10, no. 6, s. 350-355.

[10] Elias, H.: Perlin noise [online]. 7.12.1998 [cit. 4.12.2009]. Dostupné z WWW: <http://freespace.virgin.net/hugo.elias/models/m_perlin.htm>

[11] Perlin, K.: Ken Perlin's homepage [online]. 17.11.2009 [cit. 5.12.2009]. Dostupné z WWW: <http://mrl.nyu.edu/~perlin/>

[12] Scott, J.: Making Cellular Textures [online]. 4.6.2007 [cit. 4.12. 2009]. Dostupné z WWW: <http://www.blackpawn.com/texts/cellular/default.html>

[13] Bump Mapping [online]. [cit. 2. 1. 2010]. Dostupné z WWW : <http://www.yourdictionary.com/computer/bump-mapping>

[14] Kabát, Z.: 3D technologie, Bump Mapping [online]. 20.7.2004 [cit. 4.12.2009]. Dostupné z WWW: <http://www.svethardware.cz/art_doc-6D2AC72379CA43C6C1256ED6006B1BA2.html>

[15] Sanglard, F.: Shadow Mapping tutorial [online]. 2008 [cit. 6.4.2010]. Dostupné z WWW: <http://fabiensanglard.net/shadowmapping>

[16] Guinot, J.: Fog in GLSL [online]. 21.12.2007 [cit 6.4.2010]. Dostupné z WWW: <http://www.ozone3d.net/tutorials/glsl_fog/>

[17] Molofee, J.: NeHe Production - Bitmap Fonts. [online]. [cit. 10.5.2010]: Dostupné z WWW: <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=13>

[18] Homepage of demogroup Farbrausch [online]. 2009-2010 [cit. 10.5.2010]: Dostupné z WWW: <http://www.farbrausch.de/>

[19] Homepage of demogroup RGBA [online]. 2004-2007 [cit 10.5.2010]: Dostupné z WWW: <http://rgba.org/>

[20] Quilez, I.: Repository tutorials, articles and code of Iñigo Quilez on computer graphics, fractals, demoscene and more. [online]. 2010 [cit. 6.4.2010]: Dostupné z WWW: <http://iquilezles.org/www/>

[21] V2 Synthesizer System by farbrausch [online]. 2000-2010 [cit 10.5.2010]: Dostupné z WWW: <http://www.pouet.net/prod.php?which=15073>

[22] Giesen, F.: kkrunchy. [online]. 6.12.2009 [cit. 6.4.2010]: Dostupné z WWW: <http://www.farbrausch.de/~fg/kkrunchy/>

[23] UPX: the Ultimate Packer for eXecutables [online]. 27.4.2010 [cit. 10.5.2010]: Dostupné z WWW: <http://upx.sourceforge.net/>

[24] Bouke P. van Eijck: The UPACK programe package [online]. 8.1.2004 [cit. 10.5.2010]: Dostupné z WWW: <http://www.crystal.chem.uu.nl/~vaneyck/upack.html>

[25] MEW 11 SE 1.2 – real exe packer [online]. [cit. 10.5.2010]: Dostupné z WWW: <http://northfox.uw.hu/down/mew11.zip>

[26] FSG (Fast Small Good) packer [online]. [cit. 10.5.2010]: Dostupné z WWW: <http://in4k.untergrund.net/packers%20droppers%20etc/xt_fsg20.zip>

Zoznam príloh

CD nosič so zdrojovými kódmi a výsledným programom.